IL LINGUAGGIO "PIATTAFORMA" PER UTILIZZARE A FONDO TUTTE LE TECNICHE DELLA PROGRAMMAZIONE MODERNA

COMPRENDERE XML

Francesco Smelzo

EDIZIONI MASTER

i libri di

COMPRENDERE XML

di Francesco Smelzo

A Martina

INDICE

Introduzione

La sintassi di XML

Gli schemi XML

XML DOM

XPATH

XSL

Riferimenti

INTRODUZIONE

Chi si sia minimamente occupato di programmazione, ma direi più in generale di informatica, negli ultimi cinque anni non può non essersi imbattuto nel fatidico XML.
Ormai non c'è quasi linguaggio, strumento applicativo o servizio che non "parli" in XML.
Tuttavia non tutti, diciamocelo francamente, hanno capito bene cosa sia e a cosa serva precisamente.
Il nome però ci dovrebbe ricordare qualcosa: HTML ovvero il formato di composizione delle pagine Web.
HTML (HyperText Markup Language) è nato infatti, nel 1993, come insieme di istruzioni che potevano essere interpretate da un *Browser* per la visualizzazione di ipertesti. È innegabile che, da allora, questo formato abbia avuto un successo esplosivo. È per questo che il W3C (il consorzio internazionale che si occupa della standardizzazione delle tecnologie internet) ha avuto un'altra bella idea: perché non inventare un formato, simile concettualmente all'HTML, che potesse avere applicazioni più estese? E proprio da qui che nasce il nostro : eXtensible Markup Language, per gli amici XML.

1.1 UN CONFRONTO: HTML E XML

Prima di tutto, a dispetto dei nomi che potrebbero trarre in inganno, dobbiamo dire che parlando di HTML e XML non indichiamo dei veri e proprio *linguaggi* come noi programmatori intendiamo, ma, più correttamente, dei *formati* nei quali viene scritto un semplice file di testo.
Tali formati prevedono che il semplice testo venga corredato da "istruzioni" poste all'interno di tag (ovvero ciò che si trova tra i simboli **< e >**) che in qualche modo diano un "significato" (una particolare formattazione o altro) al testo che racchiudono.
Si parla quindi di *formati* perchè comunque occorre che vengano interpretati per essere letti : ad esempio una pagina scritta in HTML non

avrebbe senso se poi non venisse interpretata da un *Web Browser* che la trasformi in qualcosa di comprensibile all'utente finale.
HTML e XML sono simili in alcuni concetti di base come, ad esempio, quelli di **elemento** e di **attributo**.
Prendiamo un piccolo frammento di HTML:

```
<html>
      <body>
     <p>
        <font color="red">
Questo è un testo rosso
   </font>
     </p>
      </body>
</html>
```

Gli **elementi** sono qui costituiti dai contenitori, *<p></p>*, ad esempio, è un contenitore che denota l'inizio e la fine di un paragrafo; ** è un contenitore che delimita una particolare formattazione del testo ecc.
Il **nome** dell'elemento è dato dalla parola che compare dopo il tag di apertura (<), ovvero il nome dell'elemento *<p>* sarà "**p**" il nome di *<body>* sarà **body** e così via.
Come abbiamo visto, un elemento (ad esempio *<p>*) può avere (in XML deve avere) un corrispondente tag di chiusura (ovvero *</p>*) per delimitare la sua area di influenza sul testo.
Gli **attributi** invece sono quelle coppie di nome/valore che si trovano all'interno dei *tag* di apertura degli elementi e servono per stabilire determinate proprietà all'elemento stesso.
Nel nostro esempio di HTML un attributo è **color="red"** all'interno dell'elemento **font**.
In questo caso il **nome** dell'attributo è quello che compare a sinistra del simbolo, mentre il valore compare a destra del simbolo (in XML

deve essere sempre tra virgolette).
La differenza principale tra i due formati è che mentre in HTML i nomi degli elementi e degli attributi sono tassativi in XML sono invece liberi, non ci sono vincoli (se non quelli che da il creatore del documento con gli schemi) ad usare un nome piuttosto che un altro per un elemento o un attributo.
Ciò ha, naturalmente, una ragione: mentre HTML nasce con uno scopo preciso (quello di comporre pagine web) XML no. L'utilizzo che deve essere fatto di un documento XML dipende infatti dal programma che lo deve leggere e i nomi degli elementi e degli attributi hanno un senso in questo contesto.
Questa "libertà espressiva" dell'XML ha però un rovescio della medaglia : il documento deve seguire determinate regole sintattiche, deve essere cioè *Well Formed*.

1.2 REGOLE PER UN DOCUMENTO WELL FORMED

Le regole sintattiche di un documento XML sono poche ma importanti, un documento non *Well Formed* diventa infatti inutilizzabile dai programmi che dovranno utilizzarlo.

1.2.1 Regola numero 1 – elemento radice

Il documento deve avere uno ed un solo **elemento radice** (*root element*) ad esempio:

```
<biblioteca>
    <libro>Guerra e pace</libro>
<libro>Odissea</libro>
</biblioteca>
```

Abbiamo uno e un solo elemento (**biblioteca**) come radice.
Un documento invece del tipo:

```
<libro>Guerra e pace</libro>
<libro>Odissea</libro>
```

Sarebbe errato perché ha due elementi radice.

1.2.2 Regola numero 2 – tag finali obbligatori

Quando un elemento contiene testo o altri elementi deve essere chiuso con il corrispondente tag finale. Ad esempio:

```
<libro>Guerra e pace</libro>
```

Dove l'elemento <**libro**> deve trovare corrispondenza nel tag di chiusura </**libro**> allo stesso livello.
Senza il tag di chiusura verrebbe generato un errore dai programmi che tentino di leggerlo.

1.2.3 Regola numero 3 – elementi vuoti

Se un elemento è vuoto, nel senso che non contiene testo o altri elementi, la chiusura può avvenire anche con la sintassi abbreviata: <**libro/**>. Che sarebbe equivalente a <**libro**><**/libro**>.

1.2.4 Regola numero 4 – attributi tra virgolette

Gli attributi devono essere racchiusi tra virgolette doppie (") o semplici ('). Entrambe le virgolette utilizzate devono essere dello stesso tipo (doppie o semplici).
Ad esempio, questo metodo per definire gli attributi è corretto:

```
<libro autore="Verga">I Malavoglia</libro>
```

Questo invece è errato

```
<libro autore=Verga>I Malavoglia</libro>
```

1.2.5 Regola numero 5 – elementi non sovrapposti

Gli elementi non possono essere chiusi prima di chiudere i sotto-elementi in essi contenuti.
Questo è un esempio di sintassi non valida:

```
<biblioteca>
    <narrativa>
        <libro autore="Verga">
            I Malavoglia
    </narrativa>
        </libro>
</biblioteca>
```

Perché l'elemento **narrativa** viene chiuso prima dell'elemento **libro** che inizia al suo interno.
La sintassi corretta sarebbe stata invece:

```
<biblioteca>
    <narrativa>
        <libro autore="Verga">
            I Malavoglia
        </libro>
    </narrativa>
</biblioteca>
```

1.2.6 Regola numero 6 – gli elementi sono case-sensitive

I tag di apertura e di chiusura devono avere il nome scritto con lettere dello stesso tipo (maiuscole/minuscole), ad esempio l'elemen-

to: **<libro></libro>** è corretto, mentre non lo è: **<LIBRO></libro>**.

1.3 A COSA SERVE UN DOCUMENTO WELL-FORMED

Ci si potrebbe chiedere il perché di alcune di queste regole quando, ad esempio, all'occhio umano leggere **<libro></libro>** o **<LIBRO></libro>** è sostanzialmente la stessa cosa.
XML però è un formato di file di testo che resta sì leggibile all'uomo, ma che principalmente è diretto ad essere letto ed elaborato da altri programmi.
Per capire meglio possiamo fare un test con un programma di videoscrittura che disponga della possibilità di salvare il file come XML (ad esempio Word 2003 o OpenOffice).
Apriamo, nel nostro programma di videoscrittura, un nuovo documento e scriviamo del testo, come vediamo in figura 1.

Figura 1.1: Un normale documento in Word 2003

Salviamo quindi, su disco, il documento dandogli un nome (ad esempio documento1.doc) e utilizzando il formato di file standard del programma (nel nostro caso .DOC).
Ripetiamo l'operazione salvando con nome, ma questa volta sce-

gliendo il formato di file XML (avremo quindi su disco il file documento1.xml).
Adesso su disco dovremmo avere due file: documento1.doc e documento1.xml.
Se andiamo ad aprire con un editor di testo (come notepad di Windows) il file salvato nel formato nativo di Word, documento1.doc, avremo come risultato una serie di caratteri "incomprensibili" come quelli di cui in figura 2:
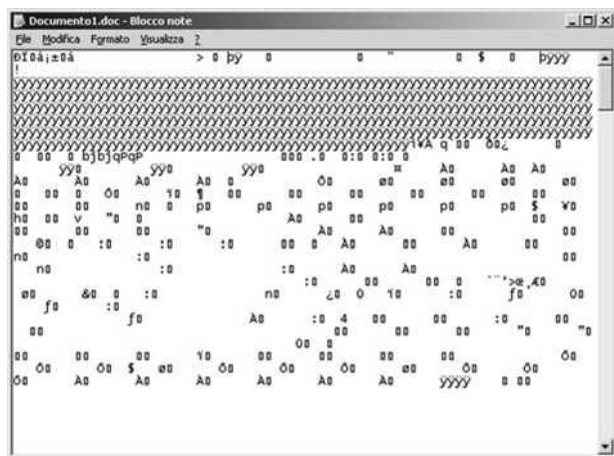
Figura 1.2: Il documento in formato .DOC in notepad

Se invece apriamo, sempre con notepad, il file XML, documento1.xml, vedremo che l'insieme risulta perfettamente leggibile (ancorché un po' "complicato") , figura 3.
Cosa significa tutto questo? Perché sebbene le origini siano tanto diverse i due documenti, aperti in Word, sono identici?
La differenza è che il formato di file standard del programma (nel nostro caso .DOC di Word) è un formato *Binario* pensato cioè per essere interpretato soltanto dal computer, mentre il formato XML è un formato di testo, che il programma *legge* e traduce in istruzioni,

ma che resta leggibile dall'uomo.

Documento1.xml - Blocco note
File Modifica Formato Visualizza ?

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml" xmlns:v=
        <w:fonts>
                <w:defaultFonts w:ascii="Times New Roman" w:fareast="Times New Roman" w
        </w:fonts>
        <w:styles>
                <w:versionOfBuiltInStylenames w:val="4"/>
                <w:latentStyles w:defLockedState="off" w:latentStyleCount="156"/>
                <w:style w:type="paragraph" w:default="on" w:styleId="Normale">
                        <w:name w:val="Normal"/>
                        <wx:uiName wx:val="Normale"/>
                        <w:rPr>
                                <wx:font wx:val="Times New Roman"/>
                                <w:sz w:val="24"/>
                                <w:sz-cs w:val="24"/>
                                <w:lang w:val="IT" w:fareast="IT" w:bidi="AR-SA"/>
                        </w:rPr>
                </w:style>
                <w:style w:type="character" w:default="on" w:styleId="Carpredefinitopar
                        <w:name w:val="Default Paragraph Font"/>
                        <wx:uiName wx:val="Car. predefinito paragrafo"/>
                        <w:semiHidden/>
                </w:style>
                <w:style w:type="table" w:default="on" w:styleId="Tabellanormale">
                        <w:name w:val="Normal Table"/>
                        <wx:uiName wx:val="Tabella normale"/>
                        <w:semiHidden/>
                        <w:rPr>
                                <wx:font wx:val="Times New Roman"/>
```

Figura 1.3: Il documento in formato .XML in notepad

Il Parser

Il programma (o il sotto-programma) che legge e interpreta il file XML è detto **Parser** (in inglese *to parse* significa "analizzare").
La ragione per cui servono quelle regole che abbiamo precedentemente enunciato risiede appunto nel fatto che al *parser* occorrono dei "punti di riferimento" certi per trasformare del semplice testo in istruzioni comprensibili alla macchina.

Formato binario o XML

Ci potrebbe ancora chiedere : ma perché Word o altri programmi simili hanno come base un formato binario e non l'XML? In realtà ci sono dei vantaggi anche nell'utilizzo dei formati binari (a parte quello, ovvio, di nascondere ai concorrenti la struttura dei documenti prodotti con il proprio programma e la possibilità di "inventare" quindi formati c.d. *proprietari*).
Il formato binario non richiede l'ulteriore operazione di parsing per essere trasformato in qualcosa di comprensibile alla macchina e quindi risulta più veloce.

Questo vantaggio, con l'accrescere della potenza dei nuovi sistemi informatici, si sta tuttavia progressivamente assottigliando ed emergono invece sempre più i pregi di un formato universale quale l'XML.
Il documento che abbiamo prodotto con Word e salvato nel formato XML, al contrario di quello salvato in formato proprietario, si può infatti leggere e interpretare con qualsiasi altro programma e anche in piattaforme del tutto diverse tra di loro.
È questo, in definitiva, il motivo del successo "esplosivo" di XML : la facilità di interscambio dei dati.

LA SINTASSI DI XML

2.1 COSTRUTTI SINTATTICI DEL DOCUMENTO XML

Come abbiamo visto le regole per scrivere un documento Well Formed sono veramente poche ed elementari.
Ad elementi, attributi e testo abbiamo già accennato, tuttavia elenchiamo qui, in maniera organica, i costrutti sintattici che può capitare di incontrare e utilizzare nella scrittura di un documento.

2.1.1 Elementi

Gli elementi sono la struttura portante di un documento XML, rappresentando veri e propri "marcatori" che fungono anche da punto di riferimento per la manipolazione attraverso programmi o fogli di stile XSL (ne parleremo più avanti).

• *Nomi degli elementi*

Ogni elemento deve avere un nome. Come abbiamo visto, deve anche rispettare alcune regole:

- il nome dell'elemento è case-sensitive
- deve iniziare con una lettera o con il simbolo di underscore (_). Ad esempio il nome di elemento che iniziasse con un numero, <1></1>, non sarebbe valido.
- un nome di elemento può contenere solo lettere, numeri, punti, caratteri di underscore (_), e trattini (-). Il carattere due punti (:) è riservato agli spazi di nomi, come vedremo in seguito.

• *Tags iniziali, finali e vuoti*

Il nome dell'elemento, come abbiamo detto, deve essere racchiuso in un Tag (in italiano : etichetta) racchiuso cioè dai simboli < e >. Se un elemento contiene altri elementi o del testo dobbiamo prevedere sia il tag di apertura che di chiusura, nel tag di chiusura dopo il sim-

bolo < deve essere inserito uno slash (/).
Un tag aperto e chiuso si presenta, ad esempio in questo modo:

```
<libro>testo</libro>
```

Quando, invece, l'elemento non contiene altri elementi o testo può essere espresso con il tag vuoto, che presenta lo slash (/) direttamente prima del simbolo >.
Quindi:

```
<libro/>
```

equivale a

```
<libro></libro>
```

• *Relazioni tra elementi*
In un documento normalmente compaiono molti elementi. Le relazioni che si stabiliscono tra di loro vengono espresse utilizzando le metafore della Famiglia o dell'Albero.
Prendiamo, ad esempio, questo documento:

```
<biblioteca>
    <reparto nome="narrativa">
        <libro>Guerra e pace</libro>
        <libro>Odissea</libro>
    </reparto>
</biblioteca>
```

La metafora della Famiglia descrive le relazioni tra gli elementi in termini di: genitore (parent), figlio (child), ascendente (ancestor), discendente (descendant), fratello (sibling).
NOTA: riportiamo i termini in inglese perché si tratta di defi-

nizioni tecniche precise che devono essere utilizzate in lingua originale.

Utilizzando la metafora della Famiglia possiamo dire che:

- **<biblioteca>** è *parent* rispetto a **<reparto>**
- **<biblioteca>** è *ancestor* rispetto a **<libro>**
- **<libro>** è *child* rispetto a **<reparto>**
- **<libro>** è *descendant* rispetto a **< biblioteca>**
- i due elementi **<libro>** sono *sibling* tra di loro

La metafora dell'albero si parla invece di radice (*root*) per definire il primo elemento, di ramo (*branch*) per definire elementi che contengono altri elementi o testo e foglia (*leaf*) per definire gli elementi vuoti o il testo contenuto in elementi.

Nel nostro esempio, secondo la metafora dell'Albero, si può dire che:

- **<biblioteca>** è la *root*
- **<reparto>** e **<libro>** sono dei rami (*branch*)
- Il testo contenuto in <libro> è la foglia (*leaf*)

2.1.2 Prologo

Si definisce prologo (prolog) quell'insieme di elementi che possono comparire prima dell'elemento radice vero e proprio del documento. Tali elementi possono essere:

- ∑ Dichiarazione
- ∑ Istruzioni di processo
- ∑ Dichiarazioni DOCTYPE
- ∑ Commenti

Riferendoci al nostro esempio precedente, riportiamo un insieme di elementi prologo, ovvero quelli che compaiono prima dell'elemento **<biblioteca>**:

dichiarazione

```
<?xml version="1.0" encoding="UTF-8"?>
```

istruzioni di processo

```
<?xml-stylesheet type="text/xsl" href="biblioteca.xsl"?>
```

dichiarazione DOCTYPE

```
<!DOCTYPE catalog SYSTEM "biblioteca.dtd">
```

commento

```
<!--aggiornato al 1/8/2006-->
<biblioteca>
    <reparto nome="narrativa">
        <libro>Guerra e pace</libro>
        <libro>Odissea</libro>
    </reparto>
</biblioteca>
```

Andiamo quindi ad analizzare più in dettaglio gli elementi del prologo.

2.1.3 Dichiarazione

La dichiarazione (*XML Declaration*) appare nella prima linea del documento XML, non è obbligatoria tuttavia, se utilizzata, deve essere posta nella prima linea del documento XML e non può essere preceduta da altro contenuto o da spazi bianchi.

- *version*

La versione (*version*) del linguaggio è l'unico attributo obbligatorio della dichiarazione. Tipicamente si indica la versione 1.0, anche se il W3C ha già rilasciato le specifiche della verione 1.1, per preservare

la compatibilità con programmi esistenti.
Questa è quindi la dichiarazione con la sola indicazione della versione:

```
<?xml version="1.0"?>
```

Nella dichiarazione, tuttavia possono comparire anche altri due elementi.

• *encoding*
Il primo è l'*encoding* ovvero la codifica utilizzata per rappresentare i caratteri nel documento.
I programmi che leggono il documento generalmente interpretano automaticamente i caratteri se utilizzano le codifiche UTF-8 o UTF-16 (che sono codifiche Unicode). È necessario quindi specificare l'*encoding* qualora si intenda utilizzare una codifica differente.
Ad esempio, per utilizzare la codifica ISO-8859-1 (comune a molte lingue occidentali) è necessario specificare la seguente dichiarazione:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

• *standalone*
Il secondo attributo facoltativo della dichiarazione è *standalone*. L'attributo *standalone* sta ad indicare se il documento contiene riferimenti a risorse esterne come schemi DTD o fogli di stile, se impostassimo come valore "yes" il programma che legge il documento potrebbe non risolvere i riferimenti esterni in esso contenuti.
Quindi, se il documento contiene riferimenti esterni, possiamo impostare la dichiarazione attribuendo a standalone il valore "no" :

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

2.1.4 Istruzioni di processo

Le istruzioni di processo (*processing instructions*) possono essere usate per includere informazioni by-passando molte delle regole della sintassi XML, possono contenere caratteri di markup (< e >) (altrimenti proibiti) e possono apparire in qualsiasi punto del documento (ma sempre dopo la dichiarazione, se prevista).
La tipica, e più utilizzata, istruzione di questo tipo è il collegamento a un foglio di stile XSL.

• *collegamento a un foglio di stile*

L'istruzione che collega il documento a un foglio di stile è utile per associare al documento XML un foglio di stile XSL o CSS che ne determini la modalità grafica di utilizzazione una volta aperto nel browser WEB (parleremo più avanti, diffusamente di XSL).
Il collegamento a un foglio di stile deve apparire nel prologo, prima di ogni elemento del documento vero e proprio.
La sintassi del collegamento è la seguente:

```
<?xml-stylesheet type="text/xsl" href="foglio.xsl"?>
```

Dove:
type – è il tipo di collegamento che può essere "text/css" per il collegamento a un foglio di stile CSS o "text/xsl" per quello ad un foglio di stile XSL.
href – è l'URI (tipicamente una URL) assoluto o relativo (rispetto a dove si trova il documento XML) del foglio di stile.
A proposito di fogli di stile collegati possiamo fare subito un semplice esempio che ci aiuti nella comprensione.
Partiamo dal nostro semplice documento di esempio che salveremo in un file chiamato biblioteca.xml:

```
<?xml version="1.0"?>
<biblioteca>
```

```
    <reparto nome="narrativa">
        <libro>Guerra e pace</libro>
        <libro>Odissea</libro>
    </reparto>
</biblioteca>
```

Se siamo in un sistema Windows apriamo il file con Internet Explorer. Il risultato sarà quello mostrato in figura
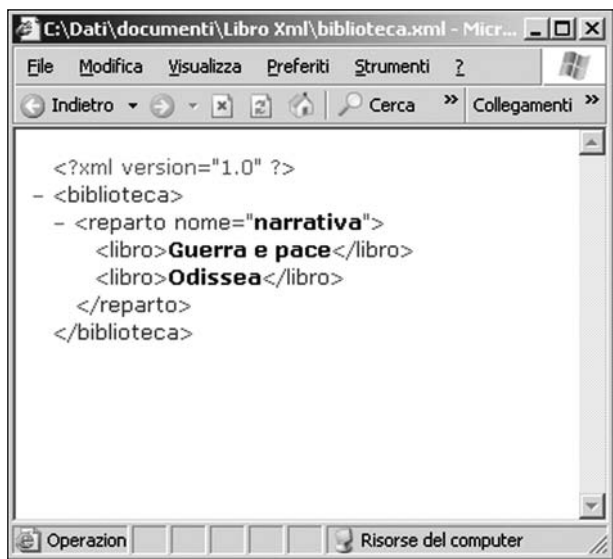
Figura 2.1: Il documento XML visualizzato in Internet Explorer

Aggiungiamo adesso la riga di collegamento a un foglio di stile CSS al documento precedente:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="biblioteca.css"?>
<biblioteca>
```

```
    <reparto nome="narrativa">
        <libro>Guerra e pace</libro>
        <libro>Odissea</libro>
    </reparto>
</biblioteca>
```

Quindi andiamo a creare, nella stessa posizione del file XML, un altro file di testo che rinomineremo in biblioteca.css (lo stesso nome che abbiamo utilizzato per il collegamento). Nel file scriviamo queste poche istruzioni che definiscono la formattazione dell'elemento **libro**:

```
libro{
    background: #EEEEEE;
margin: 2px;
    padding: 2px;
    font: bold 11px arial;
    display : block;
}
```

Riaprendo adesso il file in Internet Explorer otterremo il risultato di
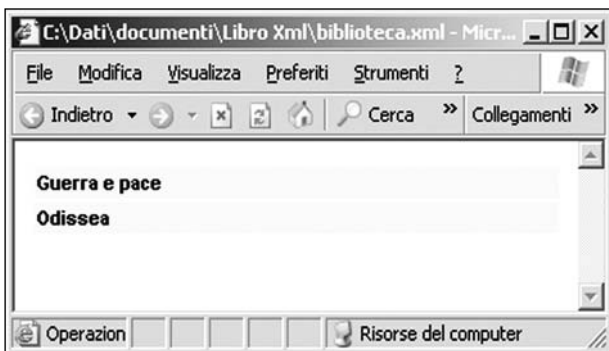
Figura 2.2: Il documento associato a un foglio di stile CSS

cui alla figura 2.
Come vedremo in seguito risultati molto più elaborati si possono ottenere utilizzando i fogli di stile XSL.

2.1.5 Dichiarazioni DOCTYPE

La dichiarazione DOCTYPE indica lo schema DTD che deve necessariamente essere applicato al documento. In questo caso elementi e attributi che possono essere inseriti devono essere quelli previsti dallo schema.
Un esempio potrebbe essere:

```
<!DOCTYPE biblioteca SYSTEM "catalog.dtd">
```

Questa dichiarazione DOCTYPE sta a indicare che il nodo radice **biblioteca** deve seguire le regole previste nello schema **catalog.dtd**.

2.1.6 Commenti

I commenti ci consentono di inserire parti di testo che non verranno considerate dai programmi che leggeranno il documento.
Essi possononо comparire in qualsiasi parte del documento (mai comunque prima della dichiarazione xml).
Il testo del commento è racchiuso, come in HTML, tra i simboli <!-- e -->.
I programmi che leggono il documento XML, interpretano --> come fine del commento, è per questo che la stringa --> non può apparire nel corpo di un commento.
Per il resto, all’interno di un commento possono apparire tutti i caratteri.
Come nei linguaggi di programmazione, i commenti contribuiscono notevolmente a migliorare la leggibilità complessiva del documento.
Il commento viene utilizzato soprattutto per inserire dei messaggi per facilitare la lettura:

```
<!-- Testo Commento -->
```

Un altro utilizzo che viene fatto del commento è quello di mascherare temporaneamente alcune parti del documento:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="biblioteca.css"?>
<biblioteca>
    <reparto nome="narrativa">
        <!--
        <libro>Guerra e pace</libro>
        -->
        <libro>Odissea</libro>
    </reparto>
</biblioteca>
```

In questo documento il primo elemento libro è stato mascherato da un commento e quindi è come se non ci fosse per il programma destinato a leggerlo.

2.1.7 Contenuto di testo

Il contenuto di testo risulta essere il testo contenuto negli elementi, è detto anche "nodo di tipo testo".
In questo frammento XML, ad esempio:

```
<libro>Guerra e pace</libro>
```

Abbiamo un elemento di nome libro che contiene un nodo di tipo testo dal valore "Guerra e pace".

2.1.8 Riferimenti a caratteri ed entità

Alcuni caratteri non possono essere utilizzati direttamente nel documento

XML perché magari rappresentano un simbolo significativo per il linguaggio.
Si prenda ad esempio il seguente documento XML :

```
<math>
    <compare>
        5 è > di 10
    </compare>
</math>
```

Apparentemente sembra tutto a posto, in realtà il documento genererebbe un errore nel Parser.

L'errore è causato dal nodo di testo : "**5 è > di 10**" in quanto contiene il carattere > che viene utilizzato in XML come marcatore.
Occorreva quindi trovare un sistema per poter scrivere nei documenti XML anche i caratteri che vengono utilizzati nella sua sintassi.
La soluzione al problema è stata di introdurre, in XML, i riferimenti.
I riferimenti non sono altro che modi alternativi di scrivere un determinato carattere.
I *riferimenti* sono di due tipi:

- *riferimenti a caratteri*
- *entità*

- <u>*riferimenti a caratteri*</u>

I riferimenti a caratteri prevedono l'inserimento del valore numerico del carattere Unicode corrispondente al carattere effettivo.
È noto infatti che i caratteri, per il computer, corrispondono in realtà a numeri. Le varie codifiche (ASCII,UNICODE ecc…) non sono altro che tabelle di corrispondenza tra un numero ed il corrispettivo carattere.
Per verificare il numero corrispondente ad un carattere nei sistemi Windows esiste una specifica utility chiamata "*Mappa caratteri*" (che, di solito, sta nel menu Programmi / Accessori.

Se apriamo *Mappa caratteri*, come in figura 3, possiamo leggere il valore numerico associato a ogni carattere.
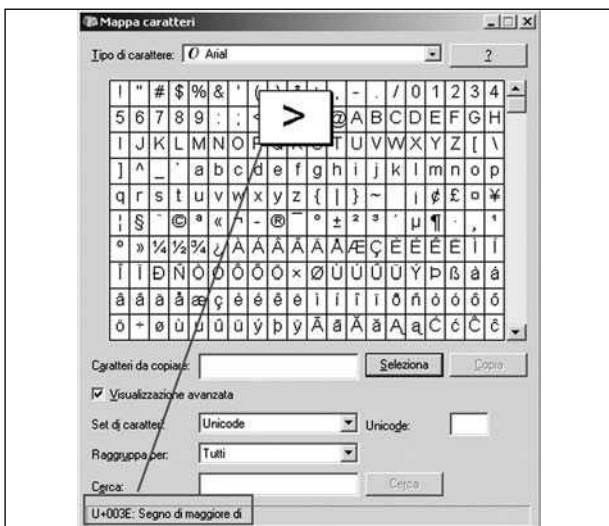
Figura 2.3: Ricaviamo il valore Unicode di un carattere in Mappa Caratteri

Nel nostro caso, per il simbolo >, leggiamo il valore U+003E; il valore effettivo è 003E espresso in formato esadecimale ovvero 62 espresso in formato decimale (chi avesse problemi per convertire un numero esadecimale in decimale può usare la calcolatrice di Windows, in modalità "scientifica": si seleziona l'opzione "hex", si scrive il numero in esadecimale e quindi si seleziona l'opzione "dec")
Nel documento XML il riferimento al carattere si ottiene :

- per i valori decimali – si racchiude il valore tra i simboli "*&*" e ";" facendo precedere al numero il simbolo "*#*". Nel nostro caso, quindi **>**;
- per i valori esadecimali - si racchiude il valore tra i simboli "*&*" e "*;*" facendo precedere al numero il simbolo "*#x*". Nel nostro caso,

quindi **>**;

Utilizzando il valore decimale, che è più semplice da ricordare, il nostro documento andrà riscritto così:

```
<math>
    <compare>
        5 è &#62; di 10
    </compare>
</math>
```

Mentre, se preferiamo l'esadecimale:

```
<math>
    <compare>
        5 è &#x003E; di 10
    </compare>
</math>
```

● *Le entità*

Utilizzare i *riferimenti* a *caratteri* non è comunque molto comodo, sia perché occorre ricavare i codici, sia perché questo metodo diminuisce la leggibilità complessiva del documento.

È per questo che ci viene in soccorso un'altra possibilità: utilizzare le *entità*.

Le *entità* sono delle "scorciatoie" da utilizzare al posto dei riferimenti numerici.

Nel nostro caso, l'*entità* corrispondente al carattere **>** è **gt** e deve essere espressa anch'essa tra i simboli "*&*" e "*;*". Quindi il nostro documento si sarebbe anche potuto scrivere così:

```
<math>
    <compare>
        5 è &gt; di 10
```

```
    </compare>
</math>
```

In XML ci sono 5 entità predefinite che rimpiazzano altrettanti caratteri che nella sintassi XML hanno un significato preciso:

Entità	Riferimento nel testo	Significato
lt	<	< (minore di)
gt	>	> (maggiore di)
amp	&	& (E commerciale)
apos	'	' (apostrofo)
quot	"	" (virgolette)

C'è tuttavia da aggiungere che, mentre i caratteri **<,>** e **&** sono vietati sempre (sia nel valore degli attributi che nei nodi di testo), l'utilizzo delle entità per i caratteri **'** e **"** è obbligatorio solo quando essi si trovino nel valore di un attributo delimitato dallo stesso tipo di carattere.
Ad esempio, questo frammento è perfettamente valido:

```
<libro autore="D'Annunzio"/>
```

Perché, è vero che nel valore dell'attributo autore, compare il carattere ' (apostrofo), ma questo non crea ambiguità nel compilatore perché come delimitatore di valore per l'attributo è stato scelto il simbolo " (virgolette).
Se come delimitatore avessimo scelto invece l'apostrofo il seguente codice sarebbe stato errato:

```
<libro autore='D'Annunzio'/>
```

E l’avremmo dovuto esprimere invece come:

```
<libro autore='D&apos;Annunzio'/>
```

Nel testo, come abbiamo detto, si possono invece usare liberamente virgolette e apostrofi,come in:

```
<libro>
    <autore>D'Annunzio</autore>
    <titolo>"Intermezzo di rime, Roma, Sommaruga, 1884"</titolo>
</libro>
```

- *Entità personalizzate*

Come abbiamo detto, in XML ci sono 5 entità predefinite, ma nulla ci vieta di implementarne di nuove.
Si utilizza, in questo caso, la sintassi DTD (Document Type Definition).
Un caso abbastanza frequente è quello del simbolo dell’euro che può essere espresso, con il riferimento a caratteri, nella forma **€ o €**; . Poiché questa forma non è né comoda a scriversi né leggibile si può invece definirla in una dichiarazione DOCTYPE associandola ad un’entità a noi più familiare (ad esempio **eur**) come in:

```
<!DOCTYPE libro [
<!ENTITY eur "&#8364;">
]>
<libro>
    <autore>D'Annunzio</autore>
    <titolo>"Intermezzo di rime, Roma, Sommaruga, 1884"</titolo>
    <prezzo>&eur; 20</prezzo>
</libro>
```

Il risultato, visualizzato con Internet Explorer , sarà quello mostrato in figura 4.

```
<!DOCTYPE libro (View Source for full doctype...)>
<libro>
  <autore>D'Annunzio</autore>
  <titolo>"Intermezzo di rime, Roma, Sommaruga, 1884"</titolo>
  <prezzo>€ 20</prezzo>
</libro>
```

Figura 2.4: Entità &eur; interpretata dal parser di Internet Explorer

Volendo è possibile utilizzare entità personalizzate per esprimere qualsiasi altro valore di testo da sostituire, come in:

```
<!DOCTYPE libro [
<!ENTITY eur "&#8364;">
<!ENTITY op "Francesco Smelzo">
]>
<libro>
      <autore>D'Annunzio</autore>
      <titolo>"Intermezzo di rime, Roma, Sommaruga, 1884"</titolo>
      <prezzo>&eur; 20</prezzo>
      <operatore>&op;</operatore>
</libro>
```

Dove, definendo l'entità **op** (che si esprime, lo ricordiamo, come &op;) associata al valore, otterremo, in Internet Explorer, il risultato

```
<!DOCTYPE libro (View Source for full doctype...)>
<libro>
  <autore>D'Annunzio</autore>
  <titolo>"Intermezzo di rime, Roma, Sommaruga, 1884"</titolo>
  <prezzo>€ 20</prezzo>
  <operatore>Francesco Smelzo</operatore>
</libro>
```

Figura 2.5: Entità personalizzate interpretate dal parser di Internet Explorer

mostrato in figura 5.
Negli esempi precedenti abbiamo utilizzato un tipo di dichiarazione DTD interna al documento, ma nella pratica più spesso sarà preferibile condividere le entità personalizzate tra più documenti XML. Sarà quindi preferibile inserire le definizioni DTD in un file separato inserito nel documento XML come riferimento (URL assoluto o relativo). Quindi, ad esempio, riscriveremo il documento creando una dichiarazione DOCTYPE come riferimento a un documento DTD esterno:

```
<!DOCTYPE libro SYSTEM "libro.dtd">
<libro>
    <autore>D'Annunzio</autore>
    <titolo>"Intermezzo di rime, Roma, Sommaruga, 1884"</titolo>
    <prezzo>&eur; 20</prezzo>
    <operatore>&op;</operatore>
</libro>
```

E poi, nel documento libro.dtd, posto nella stessa directory, scriveremo le nostre dichiarazioni:

```
<!ENTITY eur "&#8364;">
<!ENTITY op "Francesco Smelzo">
...
```

2.1.9 Sezioni CDATA

I riferimenti a caratteri e le entità sono utili per scrivere alcuni caratteri, altrimenti vietati, che compaiono in maniera occasionale nel testo.
Tuttavia vi sono dei casi in cui è necessario inserire larghe porzioni di testo che può contenere anche caratteri vietati e sarebbe oltremodo scomodo doverli sostituire con i riferimenti.
In questo caso ci vengono in aiuto le *sezioni CDATA*.
Una sezione CDATA è una parte del testo che non viene interpreta-

ta dal *Parser*, ma che dev'essere presa così com'è scritta (ma non ignorata, a differenza dei commenti).
Ad esempio:

```
<testo>
<![CDATA[
```

Qui posso scrivere qualsiasi cosa altrimenti vietata come:

```
& (e commerciale) </tag> <<<< >>>> ecc...
]]>
</testo>
```

La sezione CDATA inizia con la notazione <![CDATA[e termina con la notazione]]> e si applica solo all'interno di elementi (e non, ad esempio, nei valori degli attributi).

2.1.10 Attributi

Gli attributi permettono l'aggiunta di informazioni agli elementi sotto forma di coppie nome/valore.
Essi appaiono nei tag di apertura dell'elemento o nell'elemento vuoto come in:

```
<libro autore="Giovanni Verga" titolo="I Malavoglia">
      ...
</libro>
```

Oppure in

```
<libro autore="Giovanni Verga" titolo="I Malavoglia"/>
```

Sono vietati invece nei tag di chiusura dell'elemento.
Il valore, come abbiamo detto in precedenza, può essere racchiuso

tra "(virgolette) o '(apostrofo), generalmente si tende ad utilizzare le virgolette.
Come si può intuire, gli *attributi* sono un sistema più "conciso" di scrivere le informazioni, dovendo utilizzare solo gli elementi si sarebbe infatti dovuto scrivere:

```
<libro>
    <autore>Giovanni Verga</autore>
    <titolo>I Malavoglia</titolo>
</libro>
```

Questa osservazione ha fatto nascere, tra gli autori di XML, due scuole di pensiero tra chi tende ad utilizzare il più possibile gli attributi e chi invece preferisce gli elementi.
A favore dell'utilizzo di attributi milita il fatto che in questo modo si producono documenti più compatti e leggeri, mentre a favore degli elementi c'è una maggiore flessibilità (un elemento è infatti sempre espandibile con altri sotto-elementi mentre gli attributi no).
In generale potremmo dire che quando si è sicuri che un determinato dato non avrà sub-articolazioni conviene sempre utilizzare gli attributi, mentre nel caso contrario preferiremo utilizzare gli elementi.

2.1.11 I namespaces

Uno dei punti di forza dell'XML è quello di potersi inventare **elementi** e **attributi** dal nome autodescrittivo.
Se ad esempio formulo un documento del tipo:

```
<biblioteca>
    <reparto nome="narrativa">
        <libro>Guerra e pace</libro>
        <libro>Odissea</libro>
    </reparto>
```

```
</biblioteca>
```

Il lettore capirà immediatamente che di cosa si tratta, perché ho utilizzato quello che si può definire un sistema coerente di nomi.
Ma esaminiamo un altro caso:

```
<negozio>
    <fornitori>
        <fornitore>
            <nome>Tizio</nome>
            <indirizzo>Via Verdi, 10 Milano</indirizzo>
        </fornitore>
    </fornitori>
    <clienti>
        <cliente>
            <nome>Caio</nome>
            <indirizzo>Via Rossi, 2 Roma</indirizzo>
        </cliente>
    </clienti>
</negozio>
```

Come possiamo vedere gli elementi **nome** e **indirizzo** sono utilizzati sia nel caso dei fornitori che dei clienti, come fare allora a far capire al *parser* che un determinato elemento appartiene all'uno o all'altro contesto?
Il sistema sta nell'utilizzo degli spazi di nomi, o *namespaces*. Un namespace collega un insieme di elementi e attributi ad un "vocabolario" o "schema" che li definisce e li contraddistingue.
Concettualmente è molto semplice. Prendiamo l'esempio precedente, se io ho spazi di **nomi** differenti tra **fornitori** e clienti l'elemento **nome** dello schema fornitori sarà diverso dall'elemento nome dello schema **clienti**.
Un po' come leggere : *fornitori.nome e clienti.nome*.

2.1.12 dichiarazione di un namespace

La dichiarazione di un namespace si effettua utilizzando il particolare attributo **xmlns** a livello del nodo che si intende attribuire a quel determinato spazio di nomi. Il valore dell'attributo **xmlns** deve essere costituito da una URL o da una URI che dovrebbe garantirne l'univocità.
La dichiarazione di un namespace avviene nella forma:

```
xmlns:prefisso="url"
```

Il prefisso rappresenta la parola che contraddistingue quel determinato namespace e che viene anteposta al nome degli elementi che fanno parte di quello spazio di nomi.
Un esempio :

```
<negozio xmlns:cl="http://miosito.com/ cliente" xmlns:fn=
                                        "http://miosito.com/ fornitore">
    <fornitori>
        <fn:fornitore>
            <fn:nome>Tizio</fn:nome>
            <fn:indirizzo>Via Verdi, 10 Milano</fn:indirizzo>
        </fn:fornitore>
    </fornitori>
    <clienti>
        <cl:cliente>
            <cl:nome>Caio</cl:nome>
            <cl:indirizzo>Via Rossi, 2 Roma</cl:indirizzo>
        </cl:cliente>
    </clienti>
</negozio>
```

Si dichiarano nel documento due *namespaces* : **cl**, per gli elementi che appartengono al gruppo clienti e **fn** per quelli che appartengono al

gruppo fornitori.
Da notare che il nome del namespace viene poi anteposto a quello dell'elemento a cui appartiene seguito da segno : (due punti).

2.1.13 Ambito dei namespace

Nell'esempio precedente abbiamo dichiarato i namespaces direttamente nell'elemento radice del documento, questo perché la dichiarazione ha un ambito preciso di validità : è limitata all'elemento all'interno del quale viene dichiarata ed ai suoi discendenti.
Ovviamente, dichiarando uno spazio di nomi all'interno dell'elemento radice gli elementi appartenenti al *namespace* possono comparire in qualsiasi posizione del documento.
Viceversa potrei dichiarare il namespace solo lì dov'è effettivamente necessario, come in :

```
<negozio>
    <fornitori>
        <fn:fornitore xmlns:fn="http://miosito.com/schemi/fornitore">
            <fn:nome>Tizio</fn:nome>
            <fn:indirizzo>Via Verdi, 10 Milano</fn:indirizzo>
        </fn:fornitore>
    </fornitori>
    <clienti>
        <cl:cliente xmlns:cl="http://miosito.com/schemi/cliente">
            <cl:nome>Caio</cl:nome>
            <cl:indirizzo>Via Rossi, 2 Roma</cl:indirizzo>
        </cl:cliente>
    </clienti>
</negozio>
```

Se però, a questo punto, inserissi un elemento appartenente ad un namespace al di fuori del suo ambito di validità otterrei un errore del parser:

```
<negozio>
    <fornitori>
        <fn:fornitore xmlns:fn="http://miosito.com/schemi/fornitore">
            <fn:nome>Tizio</fn:nome>
            <fn:indirizzo>Via Verdi, 10 Milano</fn:indirizzo>
        </fn:fornitore>
        <fn:fornitore></fn:fornitore> fl ERRORE
    </fornitori>
. . .
</negozio>
```

2.1.14 Namespace di default

Negli esempi precedenti abbiamo visto come dichiarare dei namespace associati ad un prefisso. C'è però anche la possibilità di dichiarare un namespace senza prefisso, detto anche *namespace di default* (ovvero predefinito).

In questo caso tutti gli elementi scritti senza prefisso, rientrano nell'ambito di validità di quel namespace:

```
<negozio  xmlns="http://miosito.com/schemi/negozio">fl  NAMESPACE
DEFAULT
    <fornitori>
        <fn:fornitore xmlns:fn="http://miosito.com/schemi/fornitore">
            <fn:nome>Tizio</fn:nome>
            <fn:indirizzo>Via Verdi, 10 Milano</fn:indirizzo>
        </fn:fornitore>
        <fn:fornitore></fn:fornitore>
    </fornitori>
    <clienti>
        <cl:cliente xmlns:cl="http://miosito.com/schemi/cliente">
            <cl:nome>Caio</cl:nome>
            <cl:indirizzo>Via Rossi, 2 Roma</cl:indirizzo>
```

```
        </cl:cliente>
    </clienti>
</negozio>
```

In questo caso gli elementi **negozio**, **fornitori** e **clienti** che non hanno prefisso rientrano nell'ambito di validità del *namespace* predefinito.
I *namespaces* di default seguono anch'essi la regola dell'ambito di validità per cui se in un elemento interno dichiaro un altro spazio di nomi a quell'elemento ed ai suoi discendenti si applica il nuovo spazio di nomi. Questo ci consentirebbe, nel nostro caso, di definire degli spazi di nomi in modo da non dover usare prefissi:

```
<negozio xmlns="http://miosito.com/schemi/negozio">
    <fornitori>
        <fornitore xmlns="http://miosito.com/schemi/fornitore">
            <nome>Tizio</nome>
            <indirizzo>Via Verdi, 10 Milano</indirizzo>
        </fornitore>
    </fornitori>
    <clienti>
        <cliente xmlns="http://miosito.com/schemi/cliente">
            <nome>Caio</nome>
            <indirizzo>Via Rossi, 2 Roma</indirizzo>
        </cliente>
    </clienti>
</negozio>
```

Per una migliore comprensibilità del documento, tuttavia, si tende a dichiarare i namespaces (di default o con prefisso) tutti nell'elemento radice anche per evitare problemi in caso di elementi nidificati appartenenti a *namespaces* diversi.

● *Namespaces e attributi*
Negli esempi precedenti abbiamo visto i namespaces applicati soltanto agli elementi, tuttavia essi si applicano, allo stesso modo (con il prefisso davanti al nome) anche agli attributi.
Si consideri, ad esempio:

```
<libro xmlns:store="http://miosito.com/store">
    <autore>Giovanni Verga</autore>
    <titolo store:prezzo="  10">I Malavoglia</titolo>
</libro>
```

In questo caso il *namespace* **store** definisce un solo attributo, **prezzo**, e non anche l'elemento a cui appartiene.
Quando invece il *namespace* è applicato all'elemento tutti i suoi attributi e quelli dei suoi discendenti gli appartengono senza bisogno di dichiarare a loro volta il prefisso, come in:

```
<libro xmlns:store="http://miosito.com/store">
    <autore>Giovanni Verga</autore>
    <store:titolo prezzo="  10">I Malavoglia</titolo>
</libro>
```

In questo caso, infatti, sia l'elemento **titolo** che l'attributo **prezzo** appartengono allo spazio di nomi **store**.
Avrete notato che nella dichiarazione di un namespace abbiamo inserito una URL, questa URL serve da identificatore univoco (unique ID) nei confronti degli altri namespaces eventualmente presenti nel documento.
Dato che utilizziamo gli spazi di nomi semplicemente per evitare confusioni tra elementi dallo stesso nome questa URL può essere anche del tutto fittizia.

CONCLUSIONI

In questo capitolo abbiamo trattato della sintassi necessaria affinché il documento possa essere correttamente interpretato da un *parser*. Il documento, però potrebbe avere elementi e attributi disposti in qualsiasi ordine e con qualsiasi nome.
A volte invece, per i nostri scopi, è necessario che il documento sia scritto utilizzando soltanto determinati elementi e non altri ecc…
A questo servono gli *schemi* di cui parleremo nel prossimo capitolo.

GLI SCHEMI XML

3.1 VALIDAZIONE DEL DOCUMENTO

La **validazione** del documento è cosa del tutto diversa dalla sua correttezza semantica. Il documento potrebbe essere infatti perfettamente well-formed (e quindi corretto dal punto di vista delle regole dell'XML) ma non essere valido rispetto alle regole dello schema dichiarato.

Mentre le regole semantiche dell'XML si applicano a tutti i documenti XML gli schemi sono regole ulteriori definite dall'utente che "restringono" le possibilità di inserire elementi ed attributi ai criteri in essa definiti.

Per fare un esempio richiamandosi al linguaggio HTML.

Se scrivessi del codice di questo tipo:

```
<td>fl cella
    <tr>flriga
        <table></table>fltabella
    </tr>
</td>
```

Sotto il profilo dell'XML sarebbe senz'altro corretto perché rispetta tutte le regole di correttezza semantica che in precedenza abbiamo enunciato.

Come codice HTML invece questo frammento non sarebbe valido (ed il browser potrebbe interpretarlo non correttamente) perché HTML rispetta uno schema che ci dice che:

- l'elemento *<tr>* (riga di tabella) deve comparire obbligatoriamente al di sotto di un elemento *<table>*
- l'elemento *<td>* (cella di una riga)deve comparire obbligatoriamente al di sotto di un elemento *<tr>*

Cioè l'esatto contrario di quello che abbiamo fatto noi.

Lo schema, cioè, definisce una serie di regole logiche per cui una cella non può essere dichiarata prima della riga a cui appartiene, né questa può essere dichiarata prima della tabella relativa.
Ma facciamo un altro esempio, sempre rifacendosi all'HTML:

```
<body>
    <div>
        <pippo></pippo>
    </div>
</body>
```

Anche qui, codice perfettamente valido per XML ma non per HTML. Lo schema dell'HTML infatti non prevede, tra gli elementi che possono essere utilizzati l'elemento *<pippo>* per cui esso non riveste alcun significato nell'ambito del linguaggio.
In XML, lo abbiamo detto, la creazione di elementi e di attributi è libera (sempre nell'ambito delle regole della sintassi) tuttavia questa libertà può essere volutamente limitata a:

- Regole per il posizionamento degli elementi (l'elemento x non può stare all'interno dell'elemento y ecc…) o degli attributi (l'attributo z può essere presente solo all'interno dell'elemento x ecc…)
- Regole per la presenza di elementi e attributi (l'elemento x non è previsto, quello y sì ecc…)
- Regole per i valori (l'attributo z può avere soltanto valori dall'1 al 10 ecc…)
- Regole per l'unicità (l'elemento y può comparire sono una volta sotto l'elemento x)
- Regole per l'obbligatorietà (l'elemento x deve sempre essere presente)

A differenza dell'HTML però, dove lo schema era predefinito, in XML è possibile impostare un proprio schema che dev'essere rispettato da

tutti i documento che lo dichiarino.
Ma come si definisce uno schema per i documenti XML?
Gli schemi possono essere definiti attraverso tre tecniche differenti:

- **DTD** o Document Type Definition – che abbiamo già visto come uno degli elementi che possono comparire nel prologo di un documento XML
- **XDR** o XML-Data Reduced – una forma semplificata di schema che utilizza XML
- **XSD** o XML Schema definition language – che è il linguaggio standard attualmente in uso per la definizione di Schemi, anch'esso basato su sintassi XML.

Per la presente trattazione abbiamo scelto di riferirsi al linguaggio XSD perché è quello più completo ed utilizzato nella creazione di schemi per XML.

3.1.1 Quando serve uno schema

Diciamo subito che se sviluppiamo un programma che utilizza XML al suo interno come "repository" di dati, una situazione cioè dove ci "inventiamo" elementi ed attributi come meglio ci torna utile, non è necessario creare uno schema né tantomeno ricorrere alla validazione, in quanto siamo noi stessi allo stesso tempo creatori e fruitori del documento.
In una situazione, invece, in cui dovremmo pensare di scambiare il documento con altri le cose cambiano radicalmente.
Pensiamo ad uno scenario in cui dobbiamo esportare i dati da un database in una piattaforma (es. SQL Server su Window) per inviarli ad una filiale in America che li dovrà importare su un altro database in una piattaforma diversa (es. MySql su Linux).
In una situazione di questo tipo l'XML è il candidato ideale, ma come "mettersi" d'accordo sul formato da utilizzare, su come sono

mappati i campi e quale tipo di dati ospitano?
È qui che entra in gioco il ruolo degli schemi.

3.2 GLI SCHEMI XSD

Gli schemi XSD sono lo strumento che permette di definire la struttura e i tipi di dati contenuti in un documento XML.
In primo luogo c'è da dire che anche lo schema,di per sé, è un documento XML, fisicamente esso è un file di testo (spesso con estensione .XSD) che ha, più o meno un aspetto del genere:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="biblioteca">
  <xs:complexType>
   <xs:sequence>
    <xs:element maxOccurs="unbounded" name="reparto">
     <xs:complexType>
      <xs:sequence>
       <xs:element maxOccurs="unbounded" name="libro"
                                                        type="xs:string" />
      </xs:sequence>
      <xs:attribute name="nome" type="xs:string" use="required" />
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Forse l'avrete riconosciuto, sì, questo è lo schema che descrive uno dei documenti che abbiamo utilizzato come esempio:

```
<biblioteca>
```

```
    <reparto nome="narrativa">
        <libro>Guerra e pace</libro>
        <libro>Odissea</libro>
    </reparto>
</biblioteca>
```

3.2.1 L'elemento <schema>

Il documento dello schema XSD deve innanzitutto dichiarare un elemento di primo livello di nome **schema** e il riferimento al namespace *"http://www.w3.org/2001/XMLSchema"*. Il *namespace* può essere dichiarato come default o con prefisso, comunemente si assegna a questo spazio di nomi il prefisso **xs**.

Il primo elemento (o radice) di un documento XSD sarà quindi:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...
</xs:schema>
```

Oltre al namespace obbligatorio, l'elemento **schema** può dichiarare anche altri spazi di nomi, tra i quali quello del documento al quale si riferisce.
Ad esempio, l'elemento radice di uno schema si potrebbe presentare come:

```
<xs:schema
  xmlns="http://biblioteca.org/schema"
  targetNamespace="http://biblioteca.org/schema"
  elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
...
</xs:schema>
```

in questo elemento <schema> il frammento:

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

indica che gli elementi ed i tipi di dati usati nello schema appartengono al *namespace* "*http://www.w3.org/2001/XMLSchema*" e che devono comparire con il prefisso **xs**.
Il frammento:

```
targetNamespace="http://biblioteca.org/schema"
```

invece indica che gli elementi definiti dallo schema appartengono al namespace "*http://biblioteca.org/schema*".
Il frammento:

```
xmlns="http://biblioteca.org/schema"
```

stabilisce quale dovrà essere il namespace di default.
Infine:

```
elementFormDefault="qualified"
```

Sta ad indicare che ogni elemento usato all'interno del documento XML che applica questo schema deve dichiarare il namespace, ad esempio, in questo caso:

```
<biblioteca xmlns="http://biblioteca.org/schema">
...
</biblioteca>
```

3.2.2 Referenziare lo schema nel documento XML

All'interno del documento XML che aderisce ad uno schema, per garantirne la validazione, occorre referenziare lo schema stesso, come ad esempio:

```
<?xml version="1.0"?>
<biblioteca xmlns="http://biblioteca.org/schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://biblioteca.org/schema biblioteca.xsd">
  <reparto nome="narrativa">
    <libro>Guerra e pace</libro>
            <libro>Odissea</libro>
      </reparto>
</biblioteca>
```

Possiamo notare che, nell'elemento radice, abbiamo inserito vari riferimenti.
Il primo rappresenta lo spazio di nomi che è dichiarato nel **targetNamespace** dello schema:

```
xmlns="http://biblioteca.org/schema"
```

Poi abbiamo la dichiarazione dello spazio di nomi di istanza di XML Schema, ovvero:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

una volta che abbiamo a disposizione il namespace **xsi** dovremmo poi specificare la collocazione dello schema:

```
xsi:schemaLocation="http://biblioteca.org/schema biblioteca.xsd"
```

con il namespace di riferimento (in questo caso http://biblioteca.org/schema) seguito dal percorso del file dello schema preceduto da uno spazio.

3.2.3 Tipi

Gli elementi del linguaggio XML Schema usati per modellare lo sche-

ma si dividono in:

1. **Simple Types** - Elementi e attributi semplici o restrizioni
2. **Complex Types** – Elementi contenenti altri sub-elementi
3. **Data Types** – tipi di dati di base (built-in) o derivati

La combinazione di Simple Types e Complex Types descrive la struttura del documento al quale lo schema di riferisce, i Data Types sono invece utili a qualificare i valori espressi negli attributi e negli elementi con solo testo.

Simple types

I tipi semplici comprendono:

1. elementi semplici
2. attributi
3. restrizioni o facets

Elementi semplici

Un elemento semplice è un elemento XML, privo di attributi, che contiene solo testo (privo quindi di ulteriori sotto-elementi).
La sintassi di base per definire un elemento semplice è:
<xs:element name="???" type="???"></xs:element>
Dove l'attributo name rappresenta il nome dell'elemento, mentre type è il tipo di dati in esso contenuti.
XML Schema dispone, come meglio vedremo in seguito, di diversi tipi di dati predefiniti tra i più comuni:

- string – stringhe di caratteri
- decimal – numeri
- integer – numeri interi
- boolean – valori booleani
- date – data
- time – ora

quindi potremo definire, ad esempio, un elemento semplice così:

```
<xs:element  name="libro" type="xs:string"></xs:element>
```

che corrisponderebbe ad un elemento del documento XML di questo tipo:

```
<libro>Odissea</libro>
```

Gli elementi semplici possono poi avere un valore di **default** che viene assegnato quando non viene assegnato nessun valore:

```
<xs:element  name="libro" default="senza titolo"
                                    type="xs:string"></xs:element>
```

Oppure possono anche avere un valore **fixed** che viene assegnato sempre (**fixed** esclude **default** e viceversa):

```
<xs:element  name="annoCorrente" default="2006" type="xs:inte
                                             ger"></xs:element>
```

Attributi
Se un elemento contiene attributi diventa per ciò stesso un elemento complesso, gli attributi sono definiti tuttavia come Simple Types, la sintassi per definire un attributo è:

```
<xs:attribute name="???" type="???"/>
```

in pratica l'attributo di definisce allo stesso modo che un elemento semplice:

```
<xs:attribute name="nome" type="xs:string"/>
```

Corrispondente, nel documento XML, a :

```
<reparto nome="narrativa">
```

Anche l'attributo, come l'elemento semplice, può avere la definizione di **default** o **fixed** :

```
<xs:attribute name="nome" default="senza nome" type="xs:string"/>
<xs:attribute name="anno" default="2006" type="xs:integer"/>
```

Gli attributi definiti dallo schema, come impostazione predefinita, sono opzionali, cioè possono comparire o non comparire nel documento XML al quale lo schema si applica. Se si desidera invece che un attributo sia obbligatorio, cioè che debba comparire sempre, è necessario usare **use** con il valore required:

```
<xs:attribute name="nome" use="required" type="xs:string"/>
```

Restrizioni (facets)

I dati che possono essere utilizzati come testo negli elementi semplici e come valori degli attributi possono essere, come abbiamo visto, limitati ad un particolare tipo (stringa, data, intero ecc…).
Questo però potrebbe non essere sufficiente : potrebbe, ad esempio, essere necessario che una stringa sia lunga al massimo 10 caratteri, che i valori possano essere ricompresi in un certo intervallo e così via.
È per questo che ai tipi di base possono essere applicate ulteriori restrizioni, dette anche *facets*.
Una restrizione al tipo di dati contenuto in un elemento semplice può essere espressa così:

```
  <xs:element name="intervallo">
   <xs:simpleType>
```

```
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="50"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Ciò avrà come conseguenza che il seguente elemento che dovesse comparire nel documento sarebbe considerato errato dal validatore:

```
<intervallo>60</intervallo>
```

mentre questo sarebbe corretto:

```
<intervallo>40</intervallo>
```

Ma nell'esempio precedente da notare è anche il modo diverso di associare il tipo all'elemento: mentre in precedenza abbiamo visto definire il tipo con l'attributo **type** qui, trattandosi di dover definire il tipo in maniera più articolata, la definizione del tipo è contenuta nell'elemento *<xs:simpleType>* all'interno di *<xs:element>*.
In questo modo comunque saremmo costretti a definire, magari la stessa restrizione, elemento per elemento e attributo per attributo. Se una facet è comune a più elementi e attributi conviene allora definirla esternamente:

```
<xs:simpleType name="range">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="50"/>
  </xs:restriction>
</xs:simpleType>
```

ed inserirne il nome nell'attributo **type**:

```
<xs:element name="intervallo" type="range"></xs:element>
```

Le restrizioni, oltre che su valori, possono riguardare anche insiemi. Ad esempio, se vogliamo far sì che in un attributo possano apparire sono determinati valori:

```
<xs:attribute name="color">
   <xs:simpleType>
      <xs:restriction base="xs:string">
         <xs:enumeration value="red"></xs:enumeration>
         <xs:enumeration value="white"></xs:enumeration>
         <xs:enumeration value="black"></xs:enumeration>
      </xs:restriction>
   </xs:simpleType>
</xs:attribute>
```

È possibile poi definire delle restrizioni molto più sofisticate basate sui *patterns*.
Ad esempio per far sì che in un attributo compaia il prezzo espresso in dollari (con il simbolo $ seguito da spazi e numeri) possiamo usare questa definizione:

```
     <xs:attribute name="dollari">
       <xs:simpleType>
         <xs:restriction base="xs:string">
           <xs:pattern value="\$\s*\d+"></xs:pattern>
         </xs:restriction>
       </xs:simpleType>
     </xs:attribute>
```

ciò fa sì che, nel documento XML, questo attributo sia valido:

```
<elemento dollari="$ 30"/>
```

mentre questo no:

```
<elemento dollari="  30"/>
```

La formula di validazione risiede nell'elemento *<xs:pattern>* che utilizza la sintassi delle espressioni regolari.
Le espressioni regolari, o *regular expressions*, sono la tecnica di riconoscimento di un testo o parti di esso a partire da un espressione o *pattern*, esse rappresentano un vero e proprio pilastro della programmazione e, vista la vastità dell'argomento, non possono essere trattate in questa sede, in internet comunque si trovano approfonditi tutorials in merito sia sulla library di Microsoft (*http://msdn.microsoft.com/library*) che su altri siti (basta cercare su Google "regular expressions tutorial").
Un altra restrizione, comunemente usata, è quella sulla lunghezza delle stringhe.
Ad esempio, se vogliamo che la lunghezza del testo contenuto in un elemento sia di 10 caratteri:

```
<xs:element name="libro">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:length value="10"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

oppure se vogliamo che venga anche stabilito un numero minimo e massimo di caratteri:

```
<xs:element name="libro">
```

```
<xs:simpleType>
 <xs:restriction base="xs:string">
  <xs:minLength value="5"/>
  <xs:maxLength value="8"/>
 </xs:restriction>
</xs:simpleType>
</xs:element>
```

Le restrizioni utilizzabili sono riassunte in questo specchietto riepilogativo:

Restrizione	Descrizione
enumeration	Lista di valori
fractionDigits	Numero massimo di decimali
length	Numero esatto di caratteri
maxExclusive	Il valore numerico deve essere inferiore a quello specificato
maxInclusive	Il valore numerico deve essere inferiore o uguale a quello specificato
maxLength	Numero massimo di caratteri ammessi
minExclusive	Il valore numerico deve essere superiore a quello specificato
minInclusive	Il valore numerico deve essere superiore o uguale a quello specificato
minLength	Numero minimo di caratteri ammessi
pattern	Sequenza di caratteri ammessi
totalDigits	Numero di cifre ammesso
whiteSpace	Specifica come vengono gestiti gli spazi bianchi (accapo, tabs, spazi e interruzioni di riga) .

Complex types

Gli elementi complessi sono:

- Elementi che contengono altri elementi

```
<persona>
```

```
    <nome>Mario</nome>
</persona>
```

- Elementi vuoti con attributi

```
<persona nome="Mario"/>
```

- Elementi con attributi che contengono solo testo

```
<persona nome="Mario">Tel. 338 3323232</persona>
```

- Elementi che contengono altri elementi e testo

```
<persona nome="Mario">
Tel. 338 3323232
    <indirizzo>via Verdi 7 Milano</indirizzo>
</persona>
```

3.2.4 Elementi che contengono altri elementi

Vediamo come definire un elemento che contiene altri elementi. Prendiamo ad esempio l'elemento :

```
<libro>
   <autore>Omero</autore>
   <titolo>Odissea</titolo>
</libro>
```

la sua definizione come elemento complesso sarà:

```
<xs:element name="libro">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="autore" type="xs:string" />
```

```
    <xs:element name="titolo" type="xs:string" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Anche qui, come per i simple types possiamo definire il tipo separatamente:

```
 <xs:complexType name="contenutoLibro">
  <xs:sequence>
    <xs:element name="autore" type="xs:string" />
    <xs:element name="titolo" type="xs:string" />
  </xs:sequence>
 </xs:complexType>
```

per poi richiamare il nuovo tipo nella definizione di elemento:

```
<xs:element name="libro" type="contenutoLibro"></xs:element>
```

Da notare l'elemento *<xs:sequence>* che precede la definizione dei sotto-elementi e che indica che essi devono comparire nell'esatto ordine in cui vengono definiti,*<xs:sequence>* è un "indicatore", tratteremo più avanti l'argomento.

3.2.5 Elementi vuoti con attributi

Vediamo adesso come definire un elemento vuoto (cioè che non contiene testo) come:

```
<editore nome="Mondadori"/>
```

ovvero :

```
    <xs:element name="editore">
```

```
    <xs:complexType>
      <xs:attribute name="nome"></xs:attribute>
    </xs:complexType>
  </xs:element>
```

dove l'attributo viene definito all'interno di *<xs:complexType>*.

3.2.6 Elementi con attributi che contengono solo testo

Ma un elemento con attributi potrebbe contenere anche del testo come

```
<descrizione lang="IT">testo descrizione</descrizione>
```

che, in XML Schema, si traduce in:

```
  <xs:element name="descrizione">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="lang" type="xs:string"></xs:attribute>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
```

abbiamo definito il contenuto semplice con <xs:simpleContent> al quale abbiamo associato un estensione del tipo string (<xs:extension>).

3.2.7 Elementi che contengono altri elementi e testo

Vediamo infine il caso in cui, in un elemento, si ha una commistione tra testo e sotto-elementi come in:

```
<messaggio>
  Caro <cliente>Mario Rossi</cliente>,
  Siamo lieti di comunicarle che l'ordine <idOrdine>23</idOrdine>
  è stato inoltrato.
  Il responsabile
  <responsabile>Antonio Bianchi</responsabile>
</messaggio>
```

Il tutto si può definire in XML Schema come:

```
<xs:element name="messaggio">
    <xs:complexType mixed="true">
      <xs:sequence>
        <xs:element name="cliente" type="xs:string"/>
        <xs:element name="idOrdine" type="xs:integer"/>
        <xs:element name="responsabile" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
```

cioè in pratica come un normale elemento complesso ma con in più l'attributo **mixed** in *<xs:complexType>*.

Gli indicatori
Parlando di elementi complessi di definiscono indicatori quelle istruzioni che definiscono il modo in cui gli elementi devono essere utilizzati nel documento XML.
Ci sono tre tipi di indicatori:
- di **ordinamento** – che stabiliscono l'ordine nel quale devono comparire gli elementi
- di **frequenza** – che stabiliscono quante volte devono comparire gli elementi
- di **raggruppamento** – che definiscono dei gruppi di elementi o

attributi.

3.2.8 Indicatori di ordinamento

Gli indicatori di ordinamento appaiono all'interno della dichiarazione di <xs:complexType> per stabilire l'ordine degli elementi al loro interno e, in alternativa, sono:

- all
- sequence
- choice

L'indicatore **all** indica che gli elementi possono apparire in qualsiasi ordine.
Se definiamo un complex type utilizzando **all**:

```
<xs:complexType name="contenutoLibro">
  <xs:all>
    <xs:element maxOccurs="1" name="autore" type="xs:string" />
    <xs:element maxOccurs="1" name="titolo" type="xs:string" />
  </xs:all>
</xs:complexType>
```

nel documento XML potremmo avere:

```
<libro>
  <autore>Omero</autore>
  <titolo>Odissea</titolo>
</libro>
```

ma anche:

```
<libro>
  <titolo>Odissea</titolo>
  <autore>Omero</autore>
</libro>
```

L’indicatore **sequence** indica invece che gli elementi devono comparire nell’esatta sequenza.
Se definiamo un complex type utilizzando **sequence**:

```
<xs:complexType name="contenutoLibro">
  <xs:sequence>
    <xs:element  maxOccurs="1" name="autore" type="xs:string" />
    <xs:element  maxOccurs="1" name="titolo" type="xs:string" />
  </xs:sequence>
</xs:complexType>
```

nel documento XML potremmo avere:

```
<libro>
  <autore>Omero</autore>
  <titolo>Odissea</titolo>
</libro>
```

ma non anche :

```
<libro>
  <titolo>Odissea</titolo>
  <autore>Omero</autore>
</libro>
```

L’indicatore **choice** indica che i vari elementi sono alternativi tra di loro.
Se definiamo un complex type utilizzando **choice**:

```
<xs:complexType name="contenutoLibro">
  <xs:choice>
    <xs:element  name="autore" type="xs:string" />
    <xs:element  name="titolo" type="xs:string" />
```

```
  </xs:choice>
 </xs:complexType>
```

nel documento XML potremmo avere:

```
<libro>
 <titolo>Odissea</titolo>
</libro>
oppure
<libro>
 <autore>Omero</autore>
</libro>
ma non:
<libro>
 <autore>Omero</autore>
 <titolo>Odissea</titolo>
</libro>
```

3.2.9 Indicatori di frequenza

Gli indicatori di frequenza sono:
minOccurs – ovvero il numero minimo di volte in cui un elemento può comparire (0 per gli elementi opzionali)
maxOccurs – ovvero il numero massimo di volte in cui un elemento può comparire (*unbounded* per gli elementi senza limite
Ad esempio, questo elemento può comparire 1 volta o nessuna:

```
<xs:element maxOccurs="1" minOccurs="0" name="autore"
                                          type="xs:string" />
```

In quest'altro modo invece non ci sono limiti:

```
<xs:element maxOccurs="unbounded" name="autore" type="xs:string"
```

```
/>
```

Gli indicatori di frequenza si possono avere anche all'interno di indicatori di ordinamento, cosicché in questa definizione:

```
<xs:element name="libro">
 <xs:complexType>
  <xs:all maxOccurs="1">
   <xs:element name="autore" type="xs:string" />
   <xs:element name="titolo" type="xs:string" />
  </xs:all>
 </xs:complexType>
</xs:element>
```

gli elementi autore e titolo appaiono una e una sola volta all'interno dell'elemento, dichiarando l'indicatore **maxOccurs** a livello di *<xs:all>*.

3.2.10 Indicatori di raggruppamento

Gli indicatori di raggruppamento definiscono dei gruppi di elementi o di attributi che possono poi essere utilizzati con riferimento al gruppo.
Abbiamo quindi:

- **group** – che definisce un gruppo di elementi
- **attributeGroup** – che definisce un gruppo di attributi

Per comprendere meglio prendiamo il seguente frammento XML:

```
...
 <impiegato>
  <nome>Mario</nome>
```

```
  <cognome>Rossi</cognome>
 </impiegato>
 <cliente>
  <nome>Antonio</nome>
  <cognome>Bianchi</cognome>
 </cliente>
...
```

notiamo che sia in <impiegato> che in <cliente> ci sono gli stessi elementi <nome> e <cognome>, è possibile quindi definire il gruppo come:

```
 <xs:group name="anagrafica">
  <xs:sequence>
   <xs:element name="nome" type="xs:string" />
   <xs:element name="cognome" type="xs:string" />
  </xs:sequence>
 </xs:group>
```

per poi riutilizzarlo, come riferimento, nella definizione del tipo di elemento:

```
<xs:element maxOccurs="unbounded" name="impiegato">
 <xs:complexType>
  <xs:group ref="anagrafica"></xs:group>
 </xs:complexType>
</xs:element>
<xs:element maxOccurs="unbounded" name="cliente">
 <xs:complexType>
  <xs:group ref="anagrafica"></xs:group>
 </xs:complexType>
</xs:element>
```

Per gli attributi è la stessa cosa, se abbiamo attributi ricorrenti:

```
...
 <impiegato nome="Mario" cognome="Rossi"/>
 <cliente nome="Antonio" cognome="Bianchi"/>
...
```

definiamo il gruppo con:

```
 <xs:attributeGroup name="anagrafica">
   <xs:attribute name="nome" type="xs:string" />
   <xs:attribute name="cognome" type="xs:string" />
 </xs:attributeGroup>
```

e applichiamo il riferimento con:

```
<xs:element maxOccurs="unbounded" name="impiegato">
 <xs:complexType>
  <xs:attributeGroup ref="anagrafica">
   </xs:attributeGroup>
 </xs:complexType>
</xs:element>
<xs:element maxOccurs="unbounded" name="cliente">
 <xs:complexType>
  <xs:attributeGroup ref="anagrafica"></xs:attributeGroup>
 </xs:complexType>
</xs:element>
```

I tipi indefiniti

La sintassi che abbiamo visto finora permette di definire dettagliatamente nome, posizione e caratteristiche degli elementi e degli attributi di un documento XML.
A volte, tuttavia, in un documento XML ci potrebbero essere ele-

menti o attributi in più rispetto al necessario, se questi elementi o attributi non sono definiti nello schema la validazione darà esito negativo.
Prendiamo l'esempio l'elemento *<libro>* della nostra biblioteca:

```
<libro>
  <titolo>Odissea</titolo>
  <autore>Omero</autore>
</libro>
```

per esso definiremmo lo schema:

```
<xs:element name="libro">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="autore" type="xs:string" />
   <xs:element name="titolo" type="xs:string" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

ebbene, mettiamo il caso che un altro sviluppatore utilizzi, per l'applicazione B, lo stesso formato XML che abbiamo sviluppato noi per l'applicazione A, ma per altre esigenze abbia avuto necessità di aggiungere altre informazioni a <libro> in modo da produrre un elemento di questo genere:

```
<libro>
  <titolo>Odissea</titolo>
  <autore>Omero</autore>
   <ISBN>88-430-3841-9</ISBN>
   <prezzo>  10,20</prezzo>
</libro>
```

il risultato sarebbe che l'applicazione A non potrebbe più accettare i documenti prodotti dall'applicazione B in quanto non validi secondo lo schema.
Certamente potremmo ridefinire il nostro schema per aggiungere anche i nuovi elementi, ma che succederebbe se, dopo l'applicazione B ci fosse anche un'applicazione C,D ecc… ?
In questi casi risulta più opportuno ricorrere ai **tipi indefiniti : any** per gli elementi e **anyAttribute** per gli attributi.
Aggiungendo **any** l'elemento potrà avere qualsiasi sotto-elemento oltre a quelli da noi definiti, quindi definendo l'elemento <libro> con lo schema:

```
<xs:element name="libro">
 <xs:complexType>
   <xs:sequence maxOccurs="1">
     <xs:element name="autore" type="xs:string" />
     <xs:element name="titolo" type="xs:string" />
     <xs:any minOccurs="0" maxOccurs="unbounded"
                                  processContents="skip"/>
   </xs:sequence>
 </xs:complexType>
</xs:element>
```

esso accetterà qualsiasi altro sotto-elemento presente.

Da notare la direttiva **processContents** che permette di definire in che modo il validatore dovrà interpretare i nuovi elementi che incontra e che può assumere i valori:

- **skip** – il validatore non considera i nuovi elementi
- **lax** - il validatore tenta di validare i nuovi elementi utilizzando lo schema corrente e in caso in cui non venga trovata nessuna definizione li tralascia.
- **strict** - il validatore tenta di validare i nuovi elementi utilizzando

lo schema corrente e in caso in cui non venga trovata nessuna definizione restituisce errore.

Lo stesso principio si applica agli attributi. Le altre applicazioni potrebbero infatti usare gli attributi anziché gli elementi per estendere il formato XML, come in:

```
<libro ISBN="88-430-3841-9">
  <autore>Omero</autore>
  <titolo>Odissea</titolo>
  <prezzo>  10,20</prezzo>
</libro>
```

dove si aggiunge sia un attributo che un elemento non previsti. Nello schema si può utilizzare quindi sia il tipo indefinito di elemento, **any**, che di attributo, **anyAttribute**, in questo modo:

```
<xs:element name="libro">
 <xs:complexType>
  <xs:sequence maxOccurs="1">
   <xs:element name="autore" type="xs:string" />
   <xs:element name="titolo" type="xs:string" />
   <xs:any minOccurs="0" maxOccurs="unbounded"
                              processContents="skip"/>
  </xs:sequence>
  <xs:anyAttribute processContents="skip"/>
 </xs:complexType>
</xs:element>
```

in questo modo l'estendibilità futura viene garantita.

Elementi sostituibili

Potrebbe capitare anche che, a seconda della lingua, di possano uti-

lizzare, nel documento XML, elementi dal nome diverso, ma che hanno lo stesso significato. Cioè l'elemento che in italiano era:

```
<libro>
  <autore>Omero</autore>
  <titolo>Odissea</titolo>
</libro>
```

in inglese diventi:

```
<book>
  <author>Omero</author>
  <title>Odissea</title>
</book>
```

Per fortuna, nello schema XML è possibile anche stabilire l'equivalenza tra due o più elementi.
Per impostare tale equivalenza occorre prima definire gli elementi

```
"master" sotto l'elemento radice <xs:schema> :

<xs:element name="autore" type="xs:string" />
<xs:element name="titolo" type="xs:string" />
<xs:element name="libro">
  <xs:complexType>
    <xs:sequence maxOccurs="1">
      <xs:element ref="autore" />
      <xs:element ref="titolo" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

poi si definiscono le equivalenze con **substitutionGroup** che fa ri-

ferimento al nome dell'elemento "master":

```
<xs:element name="book" substitutionGroup="libro"/>
<xs:element name="author" substitutionGroup="autore"/>
<xs:element name="title" substitutionGroup="titolo"/>
```

infine, nella definizione dell'elemento superiore a *<libro>* (*<reparto>*) si riporta il riferimento alla definizione di *<libro>*:

```
    <xs:element maxOccurs="unbounded" name="reparto">
      <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
          <xs:element ref="libro"/>
        </xs:sequence>
        ...
      </xs:complexType>
    </xs:element>
```

3.2.11 I tipi di dati di base

I tipi di dati di base in XSD si distinguono, a loro volta, tra predefiniti, o built-in, e derivati (in quanto derivano dai predefiniti).

Tipi predefiniti

I tipi predefiniti sono:

Nome tipo di dati	Descrizione
string	Stringhe di caratteri
boolean	Valori Boolean,che sono espressi con true o false.
decimal	Numeri di tipo Decimal.
float	Numeri a virgola mobile e precisione singola a 32-bit.
double	Numeri a virgola mobile e precisione doppia a 64-bit.
duration	Intervallo di tempo.
dateTime	Data completa di ora.

Time	Ora, minuti e secondi.
Date	Data
gYearMonth	Rappresenta un mese dell'anno secondo il calendario
	Gregoriano..
float	Numeri a virgola mobile e precisione singola a 32-bit.
double	Numeri a virgola mobile e precisione doppia a 64-bit.
gYear	Rappresenta un anno del calendario Gregoriano.
gMonthDay	Rappresenta una data ricorrente secondo il calendario Gregoriano, ad esempio il 4 aprile di ogni anno ecc...
gDay	Rappresenta un giorno ricorrente secondo il calendario Gregoriano, ad esempio il 4 di ogni mese.
gMonth	Rappresenta un mese ricorrente secondo il calendario Gregoriano, ad esempio giugno di ogni anno.
hexBinary	Rappresenta un dato binario secondo la codifica esadecimale.
base64Binary	Rappresenta un dato binario secondo la codifica Base64.
anyURI	Rappresenta una URI come definita dallo standard RFC 2396..
QName	Rappresenta un nome qualificato (qualified name). Un nome qualificato è composto da un prefisso e dal nome separate dal carattere di due punti (:). Il prefisso deve essere associato a un namespace, usando la dichiarazione di namespace.
NOTATION	Rappresenta un tipo di attributo NOTATION.

Da questi tipi ne derivano altri (sempre definiti dal linguaggio) che sono detti tipi di dati XML *derivati*.

Tipi derivati

In particolare, dal tipo string derivano:

Nome tipo di dati	Descrizione
normalizedString	Rappresenta una stringa "normalizzata" cioè privata degli spazi bianchi in eccesso (ad esempio " la casa" diventa "la casa"). Il tipo è derivato da string.
Token	Rappresenta una stringa tokenizzata (un token è un blocco di testo categorizzato come ad esempio le espressioni che fanno parte della sintassi di un linguaggio di programmazione). Il tipo è derivato da normalizedString.
Language	Rappresenta un identificatore di linguaggio (definito in RFC 1766). Il tipo è derivato da token.
IDREFS	Rappresenta il tipo attributo IDREFS. Contiene un insieme di valori di tipo IDREF.
ENTITIES	Rappresenta il tipo attributo ENTITIES. Contiene un insieme di valori di tipo ENTITY.
NMTOKEN	Rappresenta il tipo attributo NMTOKEN. Un NMTOKEN è l'insieme di caratteri di un nome (lettere, numeri e altri caratteri) in qualsiasi combinazione. Diversamente da Name e NCName, NMTOKEN non ha restrizioni nel carattere iniziale. Il tipo è derivato da token.
NMTOKENS	Rappresenta il tipo attributo NMTOKENS. Contiene un insieme di valori di tipo NMTOKEN.
Name	Rappresenta i nomi in XML. Un Name è un token che inizia con una lettera, un underscore, o due punti e continua con i caratteri del nome (lettere, numeri e altri caratteri). Il tipo è derivato da token.
NCName	Rappresenta nomi che non possono iniziare con il carattere dei due punti. Il tipo è derivato da Name.

ID	Rappresenta l'attributo ID definito in XML 1.0. ID dev'essere un NCName e dev'essere unico in un documento XML. Il tipo è derivato da NCName.
IDREF	Rappresenta un riferimento a un elemento che ha un attributo ID identico all'ID specificato. Un IDREF dev'essere un NCName e dev'essere un valore di un elemento o di un attributo di tipo ID presente nel documento XML. Il tipo è derivato da NCName.
ENTITY	Rappresenta il tipo attributo ENTITY in XML 1.0. Il tipo è derivato da NCName.

Il primo elemento del nostro schema XSD sarà quindi <xs:schema>

Nome tipo di dati	**Descrizione**
integer	Rappresenta una sequenza di numeri decimali con un segno iniziale opzionale(+ o -). Il tipo è derivato da decimal.
nonPositiveInteger	Rappresenta un integer minore o uguale a zero. Un nonPositiveInteger è composto da un segno negativo (-) e una sequenza di numeri decimali. Il tipo è derivato da integer.
negativeInteger	Rappresenta un integer minore di zero. È da un segno negativo (-) e una sequenza di numeri decimali. Il tipo è derivato da nonPositiveInteger.
long	Rappresenta un integer con un valore minimo di -9223372036854775808 e massimo di 9223372036854775807. Il tipo è derivato da integer.
int	Rappresenta un integer con un valore minimo di -2147483648 e massimo di 2147483647. Il tipo è derivato da long.
short	Rappresenta un integer con un valore minimo di -32768 e massimo di 32767. Il tipo è derivato da int.

byte	Rappresenta un integer con un valore minimo di -128 e massimo di 127. Il tipo è derivato da short.
nonNegativeInteger	Rappresenta un integer maggiore o uguale a zero. Il tipo è derivato da integer.
unsignedLong	Rappresenta un integer con un valore minimo di zero e massimo di 18446744073709551615. Il tipo è derivato da nonNegativeInteger.
unsignedInt	Rappresenta un integer con un valore minimo di zero e massimo di 4294967295. Il tipo è derivato da unsignedLong.
unsignedShort	Rappresenta un integer con un valore minimo di zero e massimo di 65535. Il tipo è derivato da unsignedInt.
unsignedByte	Rappresenta un integer con un valore minimo di zero e massimo di 255. Il tipo è derivato da unsignedShort.
positiveInteger	Rappresenta un integer maggiore di zero. Il tipo è derivato da nonNegativeInteger.

Abbiamo riportato qui tutti i tipi di dati di base (predefiniti e derivati), ma naturalmente nei nostri schemi useremo più spesso solo i principali.

3.2.12 Mettiamo tutto insieme

È arrivato finalmente il momento di applicare le regole fin qui apprese per definire il nostro primo schema di documento XML.
Prendiamo quindi a modello l'esempio di documento XML a cui ci siamo riferiti più volte:

```
<?xml version="1.0"?>
<biblioteca xmlns="http://biblioteca.org/schema">
 <reparto nome="Classici">
  <libro>
```

```
      <autore>Omero</autore>
      <titolo>Odissea</titolo>
    </libro>
    <libro>
      <autore>Omero</autore>
      <titolo>Iliade</titolo>
    </libro>
  </reparto>
  <reparto nome="Fantasy">
    <libro>
      <autore>J.R.R. Tolkien</autore>
      <titolo>Il signore degli Anelli</titolo>
      <editore>Mondadori</editore>
    </libro>
  </reparto>
</biblioteca>
```

vediamo quindi quali sono le regole di schema che possiamo dedurre dal documento

Elemento/attributo	Tipo	Definizione

<biblioteca>	Elemento radice	`<xs:element name="biblioteca">`
		`  <xs:complexType>`
		`    <xs:sequence>`
		`      ... altri elem.`
		`    </xs:sequence>`
		`  </xs:complexType>`
		`</xs:element>`

<reparto>	Elemento complesso con attributi	`<xs:element name="reparto">`
		`  <xs:complexType>`
		`    <xs:sequence maxOccurs="unbounded">`

		... altri elem. </xs:sequence> ... attributo </xs:complexType> </xs:element>
nome	Attributo di <reparto>	<xs:attribute name="nome" type="xs:string"/>
<libro>	Elemento complesso	<xs:element name="libro"> <xs:complexType> <xs:sequence maxOccurs="unbounded"> ... altri elem. </xs:sequence> </xs:complexType> </xs:element>
<autore>	Elemento semplice	<xs:element name="autore" type="xs:string"/>
<titolo>	Elemento semplice	<xs:element name="titolo" type="xs:string"/>
<editore>	Elemento semplice opzionale	<xs:element minOccurs="0" name="editore" type="xs:string"/>

Il primo elemento del nostro schema XSD sarà quindi *<xs:schema>* con i relativi attributi e spazi di nomi:

```
<xs:schema  targetNamespace="http://biblioteca.org/schema"
xmlns="http://biblioteca.org/schema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
...
</xs:schema>
```

Le definizioni dei tipi vanno all'interno di *<xs:schema>*.
Un metodo abbastanza comune è quello di "nidificarle", cioè porle nella stessa relazione che hanno nel documento XML che dovrebbero rappresentare, quindi :

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema
  targetNamespace="http://biblioteca.org/schema"
  xmlns="http://biblioteca.org/schema"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  >
 <!--biblioteca-->
 <xs:element name="biblioteca" >
  <xs:complexType>
   <xs:sequence>
    <!--reparto-->
    <xs:element maxOccurs="unbounded" name="reparto">
     <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
       <!--libro-->
       <xs:element name="libro">
        <xs:complexType>
         <xs:sequence maxOccurs="unbounded">
          <!--autore-->
         <xs:element name="autore" type="xs:string"/>
          <!--titolo-->
          <xs:element name="titolo" type="xs:string"/>
          <!--editore-->
          <xs:element minOccurs="0" name="editore"
                                              type="xs:string"/>
         </xs:sequence>
        </xs:complexType>
```

```
              </xs:element>
            </xs:sequence>
            <!--attributo nome di reparto-->
            <xs:attribute name="nome" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

un metodo alternativo è quello di definire i tipi sotto l'elemento *<xs:schema>* e di riferirsi poi ad essi con l'attributo **ref**:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema
    targetNamespace="http://biblioteca.org/schema"
    xmlns="http://biblioteca.org/schema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    >
  <!--autore-->
  <xs:element name="autore" type="xs:string"/>
  <!--titolo-->
  <xs:element name="titolo" type="xs:string"/>
  <!--editore-->
  <xs:element  name="editore" type="xs:string"/>
  <!--libro-->
  <xs:element name="libro">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="autore"/>
        <xs:element ref="titolo"/>
```

```
        <xs:element minOccurs="0" ref="editore"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!--attributo nome di reparto-->
  <xs:attribute name="nome" type="xs:string"/>
  <!--reparto-->
  <xs:element name="reparto">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="libro"/>
      </xs:sequence>
      <xs:attribute ref="nome"/>
    </xs:complexType>
  </xs:element>
  <!--biblioteca-->
  <xs:element name="biblioteca" >
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded" >
        <xs:element ref="reparto"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

CONCLUSIONI

Il linguaggio XML Schema (XSD) è uno strumento potente e versatile per definire lo schema del documento XML per essere sicuri che esso sia conforme ad un determinato standard.
Anche se vi può capitare raramente di dover elaborare un vostro schema per i documenti XML sui quali lavorate, la sua comprensio-

ne è utile per comprendere schemi prodotti da altri.
Consideriamo poi che XML Schema in molti ambienti di programmazione (ad esempio nel .NET framework) è diventato un vero e proprio strumento di modellazione delle classi e la base per la serializzazione degli oggetti in formato XML.
Pur non essendo decisamente il più semplice da comprendere tra gli standards XML vale quindi la pena di fare qualche sforzo.
Ma oltre le regole sintattiche e la validazione per XML è importante un'altra attività, senza la quale il linguaggio in sé non avrebbe molta utilità pratica: il *parsing* ovvero l'analisi e la lettura di un documento da parte di un programma (il *parser* appunto).
Il *parser* analizza e restituisce i propri risultati non in maniera arbitraria, ma seguendo le regole dettate dal DOM (Document Object Model) nel prossimo capitolo vedremo quindi il modello ad oggetti dell'XML.

XML DOM

Il DOM (Document Object Model) è un'interfaccia astratta e non legata a nessun specifico linguaggio di programmazione che rappresenta la struttura di un documento XML.
I *parser* che leggono il documento XML dovrebbero rispettare tale interfaccia per essere conformi al DOM.

4.1 L'AMBIENTE DI TEST

Fino ad ora nella trattazione abbiamo seguito una linea più neutrale possibile rispetto ai vari linguaggi di programmazione, a questo punto però si pone un problema.
In pratica abbiamo ovviamente diverse implementazioni del DOM nei vari *parsers* : Microsoft (COM e .NET), Java, Php, Python ecc…
Come fare a parlare di DOM senza fare degli esempi? E come fare degli esempi senza scegliere un linguaggio di programmazione specifico?

4.1.1 Parsing con i browser Web

Dovendo quindi scegliere, scegliamo una piattaforma di test che può essere utile a tutti i programmatori : il browser (e quindi Javascript come linguaggio).
Infatti, sia Internet Explorer che Firefox supportano (pur in maniera differente) il DOM.

Il parser di Internet Explorer

IE utilizza la libreria MSXML richiamandola come oggetto ActiveX con identificatore (ProgID) diverso a seconda della versione:

```
var xmlDoc=new ActiveXObject("Microsoft.XMLDOM")
```

dove il ProgID, anziché Microsoft.XMLDOM, potrebbe essere anche:

- *MSXML.DOMDocument*
- *MSXML2.DOMDocument*

o anche altre, più recenti, versioni che possono essere installate nella macchina.
Per caricare un documento esistente con IE il codice sarà simile a:

```
var xmlDoc=new ActiveXObject("Microsoft.XMLDOM")
xmlDoc.async=false
xmlDoc.load("documento.xml")
```

Il parser di Firefox
Mozilla/Firefox invece attiva il parser attraverso un oggetto nativo:
var xmlDoc=document.implementation.createDocument("ns","root",null)
dove:

- il primo parametro è lo spazio di nomi usato dal documento XML
- il secondo parametro è il nome dell'elemento radice del file XML
- il terzo parametro lasciato sempre a null e riservato a utilizzi futuri

Per caricare un documento esistente con Mozilla/Firefox il codice sarà simile a:

```
var xmlDoc=document.implementation.createDocument("","",null);
xmlDoc.load("note.xml");
```

Uno script per i due browser
Viste le differenti modalità di creazione dell'istanza del *parser* tra i due più diffusi browser, l'unica cosa da fare è creare un nostro script che si occupi di instanziare l'oggetto corretto e caricare nel parser un file Xml riconoscendo le differenti versioni del browser.
Creiamo quindi un file chiamato "*test.html*" e, al suo interno, inseriamo lo script:

```
<script type="text/javascript" language="javascript">
function loadXml(file) {
    var xmlDoc;
```

```
    var ie = (typeof window.ActiveXObject != 'undefined');
    var moz = !ie;
                              if       (moz)       xmlDoc       =
document.implementation.createDocument("","",null);
    else xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.async=false;
    xmlDoc.load(file);
    return  xmlDoc;
}
</script>
```

in questo modo abbiamo la possibilità di disporre di un parser per un file Xml esistente nel sito.

Un esempio reale

Ma mettiamo subito alla prova il *parser* per analizzare un file Xml reale. Riprendiamo il nostro buon vecchio file Xml che abbiamo più volte usato come esempio (che avremo salvato nella stessa directory di *test.html* con il nome *biblioteca.xml*):

```
<?xml version="1.0"?>
<biblioteca xmlns="http://biblioteca.org/schema">
  <reparto nome="Classici">
    <libro>
      <autore>Omero</autore>
      <titolo>Odissea</titolo>
    </libro>
    <libro>
      <autore>Omero</autore>
      <titolo>Iliade</titolo>
    </libro>
  </reparto>
  <reparto nome="Fantasy">
```

```
    <libro>
      <autore>J.R.R. Tolkien</autore>
      <titolo>Il signore degli Anelli</titolo>
      <editore>Mondadori</editore>
    </libro>
  </reparto>
</biblioteca>
```

adesso, nel *<body>* del file html inseriamo lo script che legge il documento XML e compone il risultato nella pagina:

```
<script type="text/javascript" language="javascript">
var doc = loadXml ("biblioteca.xml");
var reparti = doc.getElementsByTagName("reparto")
for(i=0;i<reparti.length;i++){
    var reparto = reparti[i]; //nodo <reparto>
    document.write("<h2>" + reparto.getAttribute("nome",true) +
                                                        "</h2>");
    var libri = reparto.getElementsByTagName("libro")
    for(j=0;j<libri.length;j++){
        var libro = libri[j]; //nodo <libro>
        document.write("<blockquote>")
        document.write("Titolo:" + libro.getElementsByTagName("tito
                                lo")[0].firstChild.nodeValue);
        document.write("<br>");
        document.write("Autore:" + libro.getElementsByTagName
                              ("autore")[0].firstChild.nodeValue);
        document.write("</blockquote>")
    }
}
</script>
```

carichiamo quindi la pagina *test.html* nel browser (Firefox o IE) e, se

tutto è andato bene, dovremmo vedere la lista reparti/libri di cui alla figura 1.

Figura 4.1: Il documento XML letto dal parser e elaborato con javascript

Avrete sicuramente notato che, una volta impostata la funzione che carica diversamente il parser in IE e Firefox, lo script che utilizza il *parser* per leggere il documento usa dei metodi e delle proprietà (*getElementsByTagName*, *getAttribute*, *getElementsByTagName*, *firstChild*, *nodeValue*) che non cambiano a seconda del *parser* utilizzato.

Questo avviene perché entrambi i *parser*, quello di Firefox e quello di IE, implementano la stessa interfaccia: quella del DOM XML.
Ciò garantisce che qualunque sia il *parser* e qualunque sia il linguaggio di programmazione utilizzato avremo a disposizione metodi e proprietà che si chiamano allo stesso modo.
Naturalmente ogni *parser* aggiunge metodi e proprietà che non sono previsti dallo standard DOM (e che spesso semplificano molto la vita del programmatore!), tuttavia è possibile lavorare utilizzando soltanto i metodi e proprietà standard ed essere certi che il codice funzionerà con tutti i *parser* che implementano il modello ad oggetti DOM XML.

4.1.2 Il modello ad oggetti DOM XML

Figura 4.2: Il modello ad oggetti DOM XML

I nodi

Il documento XML viene visto in DOM come una gerarchia di oggetti nodo (node).

Ogni oggetto nodo ha tre proprietà fondamentali:

- nodeType – il tipo di nodo
- nodeValue – il valore del nodo

- nodeName – il nome del nodo

Questa tabella mostra i tipi di nodi definiti dal DOM e quali tipi di nodi possono contenere:

nodeType	Descrizione	Nodi figlio
Document	Rappresenta l'intero documento	Element (solo quello radice), ProcessingInstruction, Comment, DocumentType
DocumentFragment	Rappresenta un nodo Document, che può contenere una parte di un documento	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	Lista di entità definite per il documento	Nessuno
EntityReference	Un riferimento a un entità	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Un elemento	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Un attributo	Text, EntityReference
ProcessingInstruction	Una processing instruction	Nessuno
Comment	Un commento	Nessuno
Text	Il testo contenuto (stringa di caratteri) in un elemento o attributo	Nessuno
CDATASection	Un blocco di testo che può contenere caratteri che altrimenti sarebbero interpretati come marcatori	Nessuno

Entity	Un entità	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	Una notation dichiarata nel DTD	Nessuno

Vediamo anche i vari tipi di nodi in relazione al nome e al valore di ritorno (**nodeName** e **nodeValue**):

Node type	**nodeName restituisce**	**nodeValue restituisce**
Document	#document	null
DocumentFragment	#document fragment	null
DocumentType	Il nome del doctype	null
EntityReference	Il nome del riferimento a entità	null
Element	Il nome dell'elemento	null
Attr	Il nome dell'attributo	Il valore dell'attributo
ProcessingInstruction	target	Il contenuto del nodo
Comment	#comment	Il testo di commento
Text	#text	Il contenuto del nodo
CDATASection	#cdata-section	Il contenuto del nodo
Entity	Il nome dell'entità	null
Notation	Il nome della notation	null

Nel DOM la proprietà nodeType corrisponde a costanti così nominate:

1 - ELEMENT_NODE
2 - ATTRIBUTE_NODE
3 - TEXT_NODE

4 - CDATA_SECTION_NODE
5 - ENTITY_REFERENCE_NODE
6 - ENTITY_NODE
7 - PROCESSING_INSTRUCTION_NODE
8 - COMMENT_NODE
9 - DOCUMENT_NODE
10 - DOCUMENT_TYPE_NODE
11 - DOCUMENT_FRAGMENT_NODE
12 - NOTATION_NODE

4.1.3 L'oggetto Node

L'oggetto Node è la base per tutti gli altri oggetti che ne derivano per estensione (attributi, elementi ecc…). Esso dispone quindi di proprietà e di metodi comuni anche agli oggetti derivati:

Proprietà

attributes	Un oggetto NamedNodeMap contenente tutti gli attributi di un nodo
childNodes	Un oggetto NodeList contenente tutti i nodi figlio
firstChild	Il primo nodo figlio
lastChild	L'ultimo nodo figlio
namespaceURI	L' URI del namespace di un nodo
nextSibling	Il nodo immediatamente seguente a un nodo allo stesso livello
nodeName	Il nome di un nodo
nodeType	Il tipo (costante numerica) di un nodo
nodeValue	Il valore del nodo
ownerDocument	L'oggetto Document di un nodo
parentNode	Il nodo che contiene il nodo
prefix	Il prefisso del namespace di un nodo
previousSibling	Il nodo immediatamente precedente a un nodo allo stesso livello

Oggetto Attribute (derivato da Node)

appendChild(newnode)	Aggiunge un nuovo nodo figlio a un nodo e lo restituisce
cloneNode(boolean)	Crea una copia esatta di un nodo. Se il parametro è true riproduce anche i nodi figli
hasChildNodes()	Restituisce true o false a seconda del il nodo ha
	nodi figli
insertBefore (newnode,refnode)	Inserisce un nuovo nodo (primo parametro) prima del nodo esistente (secondo parametro) e restituisce il nuovo nodo
removeChild (nodename)	Rimuove il nodo figlio specificato e lo restituisce
replaceChild (newnode,oldnode)	Sostituisce il vecchio nodo (oldnode) con il nuovo (newnode) e restituisce il vecchio nodo

Ovviamente tali proprietà e metodi avranno un senso diverso a seconda del tipo di nodo a cui si applicano. Ad esempio la proprietà attributes di un nodo di tipo Attr (attributo) restituirà **null** ecc…

Dall'oggetto Node derivano, come abbiamo detto, gli altri oggetti che rappresentano particolari tipi di nodi.

Oggetto Document (derivato da Node)
Document rappresenta il documento XML stesso.

Proprietà

doctype	Il DTD o lo Schema del documento
documentElement	L'oggetto Element che rappresenta l'elemento radice del documento

Metodi

createAttribute("name")	Crea un nuovo nodo attributo
createCDATASection("text")	Crea un nuovo nodo CDATA
createComment("text")	Crea un nuovo nodo commento
createDocumentFragment()	Crea un oggetto vuoto documentFragment
createElement("name")	Crea un nuovo nodo elemento
createEntityReference("name")	Crea un nuovo nodo entityReference
createProcessingInstruction (target,text)	Crea un nuovo nodo processingInstruction
createTextNode("text")	Crea un nuovo nodo text
createTextNode("text")	Crea un nuovo nodo text
getElementById("id")	Restituisce il nodo corrispondente a un id
getElementsByTagName("name")	Restituisce un oggetto NodeList di tutti gli oggetti Nodo che hanno il nome uguale a quello specificato.

I vari *parser* estendono ulteriormente l'oggetto Document includendo i metodi per caricare l'input XML (file, URL, stringa di testo ecc…), per salvare l'XML come output, per selezionare dei set di nodi usando XPath, per trasformare l'XML usando XSL ecc…
Tali estensioni non sono vietate dal DOM, sono semplicemente non obbligatorie.
Per utilizzare un *parser* al pieno delle sue potenzialità è quindi preferibile conoscere anche tali estensioni.

Nell'utilizzo del *parser* l'oggetto Document è il primo oggetto che si instanzia e in qualche modo rappresenta il parser stesso :

```
document.implementation.createDocument("","",null); // IN FIREFOX
```

```
new ActiveXObject("Microsoft.XMLDOM"); // IN IE
```

Oggetto Element (derivato da Node)

Element rappresenta un nodo di tipo elemento, un elemento XML ad esempio:

```
<autore>Omero</autore>
```

Se un elemento ha del testo contenuto questo viene visto come nodo figlio di tipo testo (non come **nodeValue**).

Proprietà

tagName	Nome dell'elemento (lo stesso valore di nodeName)

Metodi

getAttribute(name)	Restituisce il valore dell'attributo specificato
getAttributeNode(name)	Restituisce l'attributo specificato come oggetto Attribute
getElementsByTagName (name)	Restituisce un oggetto NodeList di tutti gli oggetti Nodo che hanno il nome uguale a quello specificato.
hasAttribute()	Restituisce true o false a seconda se l'elemento ha attributi
normalize()	Unisce tutti i nodi di Testo sottostanti in un unico nodo di Testo
removeAttribute(name)	Rimuove il valore dell'attributo specificato
removeAttributeNode (name)	Rimuove l'attributo specificato
setAttribute(name,value)	Imposta il valore dell'attributo specificato
setAttributeNode(name)	Inserisce un nuovo attributo

Oggetto Attribute (derivato da Node)
Rappresenta l'attributo di un elemento.
Ad esempio in:

```
<prezzo valore="30"/>
```

L'oggetto **attribute** sarà corrispondente a valore.

Proprietà

name	nome dell'attributo (lo stesso valore di nodeName)
specified	Restituisce true se il valore è derivato da un valore di default definito nel DTD o Schema XML.
value	Restituisce o imposta il valore dell'attributo

Altri oggetti derivati da Node
Il DOM prevede poi che altri oggetti (corrispondenti ai vari altri tipi di nodo) vengano derivati dall'oggetto Node:

- Oggetto Text
- Oggetto Comment
- Oggetto CDataSection
- Oggetto Entity
- Oggetto EntityReference
- Oggetto ProcessingInstruction
- Oggetto Notation

Non ci soffermiamo particolarmente su tali oggetti in quanto non presentano proprietà e metodi particolarmente significativi rispetto a quelli ereditati da **Node**.

4.1.4 Le liste di nodi

Prevedendo l'oggetto fondamentale **Node** era logico che il DOM implementasse anche oggetti che rappresentino l'insieme di tali oggetti.

NodeList

NodeList è una matrice contenente un insieme di oggetti **node**. Tale matrice viene restituita dai metodi di selezione previsti dall'oggetto **Node** o dai sui derivati, ciò rende possibile scorrere i nodi contenuti come qualsiasi altra matrice prevista dal linguaggio di programmazione in uso, ad esempio:

```
var doc = loadXml ("biblioteca.xml");
var reparti = doc.getElementsByTagName("reparto") //NODE LIST
for(i=0;i<reparti.length;i++){
...
}
```

Proprietà

length	Numero di nodi contenuti

Metodi

item(index)	Restituisce il nodo corrispondente all'indice specificato

NamedNodeMap

NamedNodeMap è un altro tipo di matrice che viene usata per rappresentare una lista di nodi ai quali si può accedere attraverso il nome (tipicamente gli attributi di un elemento).

Proprietà

length	Numero di nodi contenuti

Metodi

getNamedItem(name)	Restituisce il nodo corrispondente al nome specificato

setNamedItem (name)	Aggiunge (o sostituisce se presente) il nodo corrispondente al nome specificato
removeNamedItem (name)	Rimuove il nodo corrispondente al nome specificato
item(index)	Restituisce il nodo corrispondente all'indice specificato

Ad esempio:

```
var doc = loadXml ("biblioteca.xml");
var reparti = doc.getElementsByTagName("reparto") //NODE LIST
for(i=0;i<reparti.length;i++){
var reparto = reparti[i]; //NODE
var namedNodeMap= reparto.attributes;//NAMEDNODEMAP
var attr = namedNodeMap.getNamedItem("nome");//NODE ATTRIBUTE
document.write(attr.nodeValue); //VALORE ATTRIBUTO
}
```

CONCLUSIONI

Il modello ad oggetti definito da DOM XML definisce gli elementi di base (proprietà e metodi) su cui si basa ogni parser basato su di esso.
Comprendere il DOM è quindi necessario per leggere e scrivere, con un programma o uno script, dei documenti XML.
Certo che utilizzare solo il DOM per trovare informazioni all'interno di elementi e attributi può essere un lavoro molto lungo e faticoso, è per questo che, in aggiunta al DOM, molti parser mettono a disposizione dei metodi di ricerca ed estrazione dei nodi più efficienti e versatili.
Tali metodi sono basati sulla tecnologia XPATH che vedremo nel prossimo capitolo.

XPATH

5.1 COS'È XPATH

XPath (che sta per percorsi XML) è un linguaggio per la ricerca di informazioni in un documento XML. XPath viene usato per navigare attraverso gli elementi e gli attributi di un documento XML.
Per coloro che hanno familiarità con i database potremmo dire che XPath rappresenta per XML quello che la clausola WHERE con i relativi operatori rappresenta per SQL.
XPath usa come metafora descrittiva delle relazioni tra nodi di un documento XML quella della famiglia (vedi capitolo I : relazioni tra elementi).

5.1.1 Relazioni tra i nodi

Le relazioni tra i nodi possono essere :

- Parent
- Children
- Siblings
- Ancestor
- Descendants

Parent

Ogni Element (tranne quello radice) , Attribute o TextNode ha un genitore, ad esempio:

```
<mondo>
    <europa>
        <italia/>
</europa>
</mondo>
```

L'elemento **Parent** del nodo *<italia>* è *<europa>* che a sua volta ha come **Parent** *<mondo>*.

Children

I nodi di tipo Element possono avere uno o più nodi figli.
Nell'esempio precedente l'elemento *<italia>* è **Children** di *<europa>*.

Siblings

I nodi di tipo Element hanno una relazione Sibling con gli elementi di pari livello.
Nell'esempio seguente sono tra loro Siblings *<europa>* e *<asia>* così come *<italia>* e *<francia>*:

```
<mondo>
      <europa>
            <italia/>
            <francia/>
</europa>
      <asia>
            <india/>
</asia>
</mondo>
```

Ancestors

Sono Ancestors (ascendenti) di un elemento tutti gli elementi che lo precedono, fino alla radice (compresa).
Ad esempio in :

```
<mondo>
      <europa>
            <italia/>
</europa>
</mondo>
```

sono **Ancestors** di *<italia>* gli elementi *<europa>*(che è anche Pa-

rent) e *<mondo>*.

Descendants
Sono **Descendants** (discendenti) di un elemento tutti gli elementi che racchiude.
Ad esempio in

```
<mondo>
    <europa>
        <italia/>
        <francia/>
</europa>
</mondo>
```

sono **Descendants** di *<mondo>* gli elementi *<europa>*, *<italia>* e *<francia>*.

5.2 LA SINTASSI XPATH

Nel corso della trattazione della sintassi di XPath è necessario, come abbiamo fatto per il DOM, fare degli esempi concreti con un parser per comprendere meglio.
Il *parser* che rende decisamente le cose più semplici è quello di Internet Explorer in cui gli oggetti **Node** dispongono del metodo **selectNodes**, il metodo **selectNodes** :
accetta come parametro un'espressione XPath
restituisce una NodeList con i nodi rispondenti ai criteri definiti
Un esempio, richiamandosi al documento che abbiamo impostato nel capitolo precedente, è:

```
var doc = loadXml ("biblioteca.xml");
var reparti = doc.selectNodes("//reparto"); //NODE LIST
```

Anche Firefox consente di utilizzare XPath, ma non in maniera al-

trettanto immediata. Quindi, per focalizzare la nostra attenzione più sul linguaggio che sugli aspetti di programmazione, i nostri esempi in javascript riguarderanno solo IE.
Il documento XML che useremo per gli esempi è sempre *biblioteca.xml*, che abbiamo visto nel capitolo precedente.

Selezione dei nodi
XPath è espressione di un percorso (come le path del sistema operativo) in un documento XML.
Il nodo è selezionato seguendo il suo percorso. Ecco alcune delle "path expressions" più utilizzate:

Espressione	**Descrizione**
nome	Seleziona tutti gli elementi figli del nodo con il nome corrispondente a quello indicato
/	Seleziona dal nodo radice
//	Seleziona i nodi nel documento dal nodo corrente che corrisponde alla selezione indipendente dalla posizione in cui siamo
.	Seleziona il nodo corrente
..	Seleziona nodo Parent del nodo corrente
@nome	Rappresenta un attributo con il nome corrispondente a quello indicato

Esempi:

path expression	**Risultato**
/biblioteca	Tutti i nodi <biblioteca> a partire dalla radice
/biblioteca/reparto	Seleziona tutti i nodi < reparto> sotto a <biblioteca> a partire dalla radice
//autore	Seleziona tutti i nodi <autore> indipendentemente da dove si trovino

Predicati
I predicati sono usati per trovare un nodo specifico o un nodo che contiene (o ha un attributo che contiene) un dato valore.
I predicati devono essere contenuti in parentesi quadre [].

Esempi:

path expression	Risultato
/biblioteca/reparto [1]	Seleziona il primo nodo <reparto> sotto a <biblioteca>
/biblioteca/reparto [last()]	Seleziona l'ultimo nodo <reparto> sotto a <biblioteca>
//titolo[@lingua='en']	Seleziona i nodi <titolo> che hanno l'attributo "lingua" con valore "en"
//libro[titolo/@lingua= 'en']/autore	Seleziona i nodi <autore> sotto <libro> che ha l'attributo "lingua" con valore "en" nell'elemento <titolo>

Selezionare nodi non conosciuti
Dei metacaratteri (wildcards) possono essere usati al posto del nome del nodo

Espressione	Descrizione
*	Qualsiasi elemento
@*	Qualsiasi attributo
node()	Qualsiasi nodo di ogni tipo

Esempi:Selezioni multiple

path expression	Risultato
/biblioteca/*	Seleziona tutti gli elementi sotto a <biblioteca> indipendentemente dal nome
/biblioteca/reparto/@*	Seleziona tutti gli attributi di <reparto> sotto a <biblioteca> indipendentemente dal nome

Selezioni multiple
L'utilizzo dell'operatore "|" in Xpath consente selezioni in più percorsi.

Esempi:

path expression	Risultato
//libro/titolo | //libro/autore	Seleziona sia <titolo> che <autore> sotto a <libro>

5.2.1 Mettiamo insieme le regole

Come abbiamo visto le regole di XPath sono semplici e intuitive.
Al tempo stesso esse sono però un vero "asso nella manica" del programmatore XML.
Pensiamo ad esempio come potremmo fare, utilizzando solo il DOM, a trovare tutti i libri appartenti al reparto "Classici" nel documento *biblioteca.xml* che abbiamo visto nel capitolo precedente:

```
var doc = loadXml ("biblioteca.xml");
var reparti = doc.getElementsByTagName("reparto")
for(i=0;i<reparti.length;i++){
    var reparto = reparti[i]; //nodo <reparto>
    if( reparto.getAttribute("nome")=="Classici") {
        var libri = reparto.getElementsByTagName("libro")
        for(j=0;j<libri.length;j++){
            var libro = libri[j]; //nodo <libro>
            //...
        }
    }
}
```

Utilizzando invece XPath è sufficiente:

```
var doc = loadXml ("biblioteca.xml");
```

```
var libri = doc.selectNodes("//reparto[@nome='Classici']")
    for(j=0;j<libri.length;j++){
        var libro = libri[j]; //nodo <libro>
        //...
    }
```

L'espressione *//reparto[@nome='Classici']* racchiude infatti tutta la ricerca che altrimenti avremmo dovuto sviluppare nodo per nodo.

5.2.2 Ricerca per assi

Ma XPath mette a disposizione anche un metodo ulteriore di ricerca, la ricerca per Assi.
Un **Asse** definisce un Nodeset (gruppo di nodi) relativamente al nodo corrente.

Gli assi sono:
ancestor
Seleziona tutti gli ascendenti del nodo corrente

ancestor-or-self
Seleziona tutti gli ascendenti del nodo corrente ed il nodo stesso

attribute
Seleziona tutti gli attributi del nodo corrente

child
Seleziona tutti i figli del nodo corrente

descendant
Seleziona tutti i discendenti del nodo corrente

descendant-or-self
Seleziona tutti i discendenti del nodo corrente ed il nodo stesso

following
Seleziona tutti i nodi di pari livello successivi a quello corrente insieme ai loro discendenti

following-sibling
Seleziona tutti i nodi di pari livello successivi a quello corrente

namespace
Seleziona tutti i namespace del nodo corrente

parent
Seleziona il nodo **Parent** del nodo corrente

preceding
Il contrario di following : seleziona tutti i nodi di pari livello precedenti a quello corrente insieme ai loro discendenti

preceding-sibling
Il contrario di following-sibling : seleziona tutti i nodi di pari livello precendenti a quello corrente

self
Seleziona il nodo corrente

La sintassi per l'utilizzo degli assi è :

```
asse::nomenodo[predicato]
```

Ad esempio:

Esempio	Risultato
child::libro	Seleziona tutti i nodi <libro> figli di quello corrente
attribute::lingua	Seleziona l'attributo "lingua" del nodo corrente

child::*	Seleziona tutti i nodi elemento figli di quello corrente
attribute::*	Seleziona tutti gli attributi del nodo corrente
child::node()	Seleziona tutti i nodi figli di quello corrente
descendant::libro	Seleziona tutti i nodi <libro> discendenti di quello corrente
ancestor:: libro	Seleziona tutti i nodi <libro> ascendenti di quello corrente
ancestor-or-self: :libro	Seleziona tutti i nodi <libro> ascendenti di quello corrente compreso il nodo corrente

5.2.3 Gli operatori XPath

Un espressione XPath può restituire valori :

- Node-set - ovvero un insieme di nodi
- String – testo
- Boolean – vero o falso
- Number – numero

Come abbiamo visto in precedenza, per la selezione possiamo utilizzare degli operatori.

Questa è la lista completa degli operatori:

Operatore	In espressioni XSL	Descrizione	Esempio	Valore di ritorno
|	|	Trova due o più node-set	//libro | //cd	Restituisce node-set con tutti i nodi <libro> e <cd>
+	+	Somma	6 + 4	10
-	-	Sottrazione	6 - 4	2
x	x	Moltiplicazione	6 x 4	24
div	div	Divisione	8 div 4	2

Operatore	In espressioni XSL	Descrizione	Esempio	Valore di ritorno
=	=	Uguale	prezzo= 9.80	vero se il prezzo è 9,80 falso se il prezzo è 9,90
!=	!=	Non uguale	prezzo!= 9.80	vero se il prezzo è 9,90 falso se il prezzo è 9,80
<	<	Minore di	prezzo < 9.80	vero se il prezzo è 9,00 falso se il prezzo è 9,80
<=	<=	Minore o uguale a	prezzo <=9.80	vero se il prezzo è 9,00 falso se il prezzo è 9,90
>	>	Maggiore di	prezzo > 9.80	vero se il prezzo è 9,90 falso se il prezzo è 9,80
>=	>=	Maggiore o uguale a	prezzo >= 9.80	vero se il prezzo è 9,90 falso se il prezzo è 9,70
or	or	OR logico	prezzo= 9.80 or prezzo= 9.70	vero se il prezzo è 9,80 falso se il prezzo è 9,50
and	and	AND logico	prezzo > 9.00 and prezzo < 9.90	vero se il prezzo è 9,80 falso se il prezzo è 8,50
mod	mod	Modulo (resto di divisione)	5 mod 2	1

Nell'esempio riportiamo anche la forma che l'operatore assume se viene utilizzato all'interno di un foglio di stile XSL (di XSL ne parle-

remo nel prossimo capitolo) dove è necessario che simboli < e > vengano sostituiti da entità corrispondenti.
Quando si utilizzano gli operatori occorre ricordarsi che XPath ha una sintassi case-sensitive, cioè che fa differenza tra maiuscole e minuscole. Per cui l'espressione : "prezzo > 9.00 **AND** prezzo < 9.90"non funzionerebbe in quanto "AND" è scritto in maiuscolo mentre l'operatore è "and", minuscolo.

5.2.4 Funzioni XPath

Oltre agli operatori, in XPath è possibile utilizzare un ricco set di funzioni per incrementare la flessibilità e la potenza nell'estrazione dei dati.

Le funzioni si dividono per categorie:

Node-Set	Accettano un node-set come argomento, ritornano un node-set, o restituiscono informazioni su un particolare nodo all'interno di un node-set.
String	Compiono valutazioni, formattazioni, e manipolazioni su stringhe.
Boolean	Valutano l'espressione argomento per ottenere un risultato Booleano (vero o falso).
Number	Valutano l'espressione argomento per ottenere un risultato numerico.

Tutte le funzioni hanno la seguente sintassi:
nomefunzione([*espressione*])
L'espressione da passare come argomento può essere anche assente, in questo caso comunque andranno sempre utilizzate le parentesi vuote "()".

Funzioni Node-Set

count()

Restituisce il conteggio dei nodi individuati con l'espressione.
Esempio

```
count(//libro) > 3
```

position()
Restituisce la posizione del nodo corrente nella relativa lista di nodi.

last ()
Restituisce il numero corrispondente all'ultima posizione in una lista di nodi. Da utilizzare nel contesto di un predicato per estrarre l'ultimo nodo di una lista.
Esempio

```
//libro[last()]
```

name ()
Restituisce il nome del nodo passato come argomento quando è usata nella forma:

name(node-set)

oppure restituisce il nome del nodo corrente quando è usata nella forma:

name()

Altre funzioni node-set meno usate sono :

id ()-Seleziona gli elementi per il valore dell'attributo "id" (minuscolo). Ma non funziona allo stesso modo con tutti gli interpreti XPath, per cui è preferibile usare l'espressione equivalente :

```
//*[@id='valore'].
```

local-name ()– che restituisce il nome locale di un nodo senza il prefisso di namespace.
L'utilizzo è identico a **name**.

namespace-uri ()- che restituisce l' URI del namespace di un nodo.

Funzioni String

Nome funzione	Descrizione	Uso	Risultato
concat	concatena due o più stringhe passate come argomenti	concat ('abc', 'd', 'ef', 'g')	abcdefg
contains	determina se una stringa è contenuta in un'altra. Se come argomenti vengono passati riferimenti a nodi di essi viene preso il valore e interpretato come stringa	contains ('stringa','ng')	true
normalize-space	Elimina gli spazi bianchi in eccesso da una stringa o dal valore di un nodo interpretato come stringa	normalize-space ('a b')	"a b"
starts-with	Restituisce vero o falso se una stringa inizia con il valore passato come secondo argomento.	starts-with ('casa','ca')	true
string	converte un oggetto in una stringa	string(1)	"1"
string-length	Restituisce la lunghezza in caratteri di una stringa	string-length('abc')	3
substring	Restituisce la sottostringa all'interno del primo argomento iniziando dalla posizione indicata nel secondo argomento (partendo da 1) per una lunghezza uguale a quella indicata nel terzo argomento	substring ('casa',1,2)	ca
substring-after	Restituisce la sottostringa all'interno del primo argomento successiva alla stringa indicata nel secondo argomento	substring-after('10/abc','/')	abc

substring-before	Restituisce la sottostringa all'interno del primo argomento precedente alla stringa indicata nel secondo argomento	substring-before('10/abc','/')	10
translate	Restituisce la sottostringa all'interno del primo argomento sostituendo i caratteri del secondo argomento con quelli del terzo	translate('bar','b','B')	Bar

Funzioni Boolean

boolean

converte un espressione in un valore booleano (vero o falso) secondo questi criteri:

- se l'argomento è un numero restituisce **false** se è 0 altrimenti **true**.
- se l'argomento è una stringa restituisce **false** se è vuota altrimenti **true**.
- se l'argomento è un riferimento a un node-set restituisce **false** se il node-set è vuoto altrimenti **true**.

not

Restituisce **true** se l'argomento è **false** e viceversa.

Ad esempio "*not(1 > 2)*" dà **false** perché l'espressione 1>2 è true.

Altre funzioni Boolean sono :

- **false() e true()** - che restituiscono sempre **false** e **true**
- **lang(nome)** - che restituisce true o false a seconda se l'attributo xml:lang del nodo corrente corrisponda o meno a quello passato come argomento

Funzioni Number
La sintassi XPath supporta le funzioni su numeri che possono essere usate con gli operatori di confronto nelle espressioni filtro.

ceiling
Arrotonda un numero decimale all'intero superiore più prossimo.
Ad esempio : *ceiling(5.5)* restituisce 6

floor
Arrotonda un numero decimale all'intero inferiore più prossimo.
Ad esempio : *floor(5.5)* restituisce 5

round
Arrotonda un numero decimale all'intero superiore o inferiore più prossimo.
Ad esempio :
round(5.6) restituisce 6
round (5.5) restituisce 6
round (5.4) restituisce 5

number
Converte l'argomento in numero, se l'argomento non può essere convertito restituisce NaN (not a number)
Ad esempio :
number('2048') restituisce 2048
number('text') restituisce NaN

sum
Utilissima funzione che accetta un node-set come argomento, converte i valori dei nodi (se può) in numeri e restituisce la somma.
Ad esempio :
sum(//@price) restituisce la somma del valore degli attributi price presenti in tutti gli elementi.

CONCLUSIONI

La tecnologia XPath fornisce un nutrito set di strumenti per la selezione di nodi di un documento XML. Essa viene spesso utilizzata nei parser come sistema di ricerca dei nodi.
Le tecnologie collegate a XML però non finiscono qui : nel prossimo capitolo parleremo di XSL ovvero lo standard che consente di trasformare un documento XML in un output di vario tipo.

XSL

6.1 COS'È XSL

Possiamo definire l'XSL come : **un'insieme di istruzioni per trasformare un determinato input (in formato XML) in vari tipi di output (altro XML, HTML, testo ecc...).**
Queste *istruzioni* sono anch'esse scritte in formato XML. XSL infatti non è altro che una applicazione specializzata di XML, nata con lo scopo di gestire la trasformazione dei dati.

XSL si basa sulla lettura dei dati da un fonte XML attraverso il linguaggio XPath.

Le *istruzioni* XSL hanno bisogno di un motore (*engine*) che le interpreti e le trasformi nell'output desiderato. Per esemplificare meglio : pensate all'HTML, il linguaggio con cui sono costruite le pagine Web, non è forse anch'esso un set di istruzioni per il *browser*? Così come HTML ha bisogno in un *browser* per essere visualizzato come pagina Web, anche XSL ha bisogno di un *engine* per eseguire la trasformazione.
I maggiori browser, Internet Explorer e Firefox, hanno un *engine* XSL incorporato.
Quando leggono cioè un documento XML con una *processing instruction* che punta a un foglio di stile XSL eseguono la trasformazione lì definita e restituiscono a video il risultato.

6.2 APPLICARE LE TRASFORMAZIONI XSL

In questo esempio inseriamo una *processing instruction* di trasformazione in un documento XML:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="biblioteca.xsl"?>
```

```
<biblioteca>
 <reparto nome="Classici">
  <libro>
   <autore>Omero</autore>
   <titolo>Odissea</titolo>
  </libro>
  <libro>
   <autore>Omero</autore>
   <titolo>Iliade</titolo>
  </libro>
 </reparto>
 <reparto nome="Fantasy">
  <libro>
   <autore>J.R.R. Tolkien</autore>
   <titolo>Il signore degli Anelli</titolo>
   <editore>Mondadori</editore>
  </libro>
 </reparto>
</biblioteca>
```

nel file *biblioteca.xsl* aggiungiamo poi le seguenti istruzioni XSL:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" />
<xsl:template match="/">
  <html>
  <body>
   <xsl:for-each select="//reparto">
      <h2>Reparto:<xsl:value-of select="@nome"/></h2>
      <xsl:for-each select="libro">
      <blockquote>
         Autore:<xsl:value-of select="autore"/><br/>
```

```
                Titolo:<xsl:value-of select="titolo"/>
            </blockquote>
            </xsl:for-each>
        </xsl:for-each>
    </body>
    </html>
</xsl:template>
</xsl:stylesheet>
```

Apriamo quindi il documento nel browser e vedremo il risultato della trasformazione come mostrato in figura 1.

Figura 6.1: Il documento XML letto dal parser e elaborato con javascript

Questo è il metodo più semplice per ottenere una trasformazione XSL, ma è anche forse il meno utile. Infatti XSL viene più spes-

so usato lato server (con ASP,ASP.NET, Java,Php ecc…) per garantire che l'output della trasformazione venga visualizzato da tutti i browser.
Lato server con ASP e javascript potremmo ottenere lo stesso output con:

```
<%
var srcTree = new ActiveXObject("Msxml2.DOMDocument");
srcTree.async=false;
srcTree.load("biblioteca.xml");
var xsltTree= new ActiveXObject("Msxml2.DOMDocument");
xsltTree.async = false;
xsltTree.load("biblioteca.xsl");
Response.Write( srcTree.transformNode(xsltTree) );
%>
```

oppure in PHP:

```
<?
$xmlDocText=file_get_contents(realpath("biblioteca.xml"));
$xslDocText=file_get_contents(realpath("biblioteca.xsl"));
$xml = new DOMDocument;
$xml->loadXML($xmlDocText);
$xsl = new DOMDocument;
$xsl->loadXML($xslDocText);
$proc = new XSLTProcessor;
$proc->importStyleSheet($xsl);
echo($proc->transformToXML($xml));
?>
```

Tutti i linguaggi hanno strumenti per la trasformazione attraverso XSL che diventa quindi uno strumento chiave per l'elaborazione di XML in modo ancor più veloce e chiaro rispetto al solo uti-

lizzo del DOM.

6.3 LA STRUTTURA DI BASE DEL FILE XSL

Un file XSL deve obbligatoriamente presentare l'elemento **stylesheet**.

Elemento <xsl:stylesheet>

```
<xsl:stylesheet
  id = " "
  extension-element-prefixes = " "
  exclude-result-prefixes = " "
  version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...
</xsl:stylesheet>
```

L'elemento <**xsl:stylesheet**> deve essere il primo nodo del documento. In esso sono contenuti tutti gli altri elementi XSL ed ha come attributi :

1. **id** – opzionale - L'identificativo unico del nodo.
2. **extension-element-prefixes** - *opzionale* - gli spazi dei nomi, separati da spazio bianco, da usare per richiamare le funzioni degli oggetti estesi (ci torneremo sopra più avanti)
3. **exclude-result-prefixes**- *opzionale* - gli spazi dei nomi, separati da spazio bianco, utilizzati che non devono essere riportati nell'output prodotto
4. **version** - *obbligatorio* – la versione del linguaggio XSL, attualmente la "1.0"
5. **xmlns:xsl** - *obbligatorio* – lo spazio di nomi del prefisso xsl: , quello che contraddistingue gli elementi della sintassi XSL da-

gli altri utilizzati nell'output. Attualmente è sempre *http://www.w3.org/1999/XSL/Transform*

L'elemento **stylesheet**, unico per ogni file, può contenere anche altri spazi di nomi oltre a quello obbligatorio che identifica XSL, questi spazi dei nomi possono essere quelli contenuti nel documento di origine o essere dei riferimenti ad oggetti di estensione, ad esempio:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:msxsl="urn:schemas-microsoft-com:xslt"
    xmlns:fo="http://www.w3.org/1999/XSL/Format">
```

Dove si dichiarano altri due namespace: **msxsl** e **fo**.

Elemento <xsl:template>

Gli elementi *template* definiscono dei modelli di trasformazione da applicare a particolari nodi o contesti.

```
<xsl:template
 name= "nome" | match = Pattern
 priority = number
 mode = " ">
</xsl:template>
```

Per seguire un'analogia rispetto ai linguaggi di programmazione possiamo pensare ai template come ai *metodi*, cioè a delle unità riutilizzabili.

In particolare si distinguono due tipi di template:

1. **I template legati agli elementi** – che cioè vengono eseguiti ogni volta che il processore incontra quel dato elemento nel file di origine.

2. **I template nominati** – più simili alle funzioni che vengono esplicitamente richiamate.

I template legati agli elementi hanno come attributi:

- **match** – *obbligatorio* – il percorso Xpath dell'elemento a cui sono collegati.
- **mode** – *opzionale* – un nome che permette di avere più template collegati allo stesso elemento e che compiono operazioni di trasformazione diverse, richiamabili ognuno con il proprio mode.
- **priority** – *opzionale* – un valore numerico che nel caso di due o più template associati allo stesso elemento determina quello da eseguire. In realtà questo attributo ha un impiego prettamente legato alla fase di debug perché consente di "nascondere" l'esecuzione di uno o più template, un po' come commentare un pezzo di codice per non farlo eseguire.

I template nominati hanno come attributo:
- **name** – *obbligatorio* – il nome con cui sono richiamabili.

I due tipi di template per essere applicati debbono essere richiamati.

I **template legati agli elementi** vengono richiamati con l'istruzione <**xsl:apply-templates**> mentre i **template nominati** con <**xsl:call-template**>.
In realtà esiste un template che non deve essere esplicitamente invocato, il template:

```
<xsl:template match="/">
</xsl:template>
```

Cioè quello associato al pattern "/" ovvero all'intero documento

di origine, un po' come il metodo Main di un programma. Questo template viene sempre eseguito.
Ma adesso vediamo subito di focalizzare questi concetti con degli esempi concreti.
Ci basiamo, per chiarezza, su un documento XML ancor più semplice:

```
<?xml version="1.0"?>
<helloworld>
    <titolo>Hallo world</titolo>
    <messaggio>Benvenuti!</messaggio>
</helloworld>
```

Applicazione di template legati agli elementi
Scriviamo il foglio di stile Helloworld utilizzando i template collegati agli elementi:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
    <xsl:apply-templates select="helloworld"></xsl:apply-templates>
</xsl:template>
<xsl:template match="helloworld">
<html>
    <head>
    <title>Esempio Hello World</title>
    </head>
    <body>
        <xsl:apply-templates select="titolo"></xsl:apply-templates>
        <xsl:apply-templates select="messaggio"></xsl:apply-templa
                                                                tes>
    </body>
```

```
</html>
</xsl:template>
<xsl:template match="titolo">
    <h1><xsl:value-of select="."/></h1>
</xsl:template>
<xsl:template match="messaggio">
        <div>
            <i>
            <xsl:value-of select="."/>
            </i>
        </div>
</xsl:template>
</xsl:stylesheet>
```

Viene dapprima eseguito il **template match="/"** che, con l'istruzione apply-templates associa il nodo *helloworld* del documento XML al template che lo gestisce (**template match="helloworld"**), quest'ultimo a sua volta oltre a scrivere dell'output richiama l'associazione ai rispettivi template per i sottonodi *titolo* e *messaggio* che definiscono la formattazione del testo contenuto.

Un po' come dire:

1. Leggi il documento di origine e vai al nodo *helloworld*
2. Sei adesso sul nodo *helloworld* qui scrivi un po' di codice di output e vai al nodo *titolo*
3. Sei adesso sul nodo *titolo* esegui l'output definito nel template associato
4. Vai al nodo *messaggio*
5. Sei adesso sul nodo *messaggio* esegui l'output definito nel template associato
6. Sei tornato al nodo *helloworld* scrivi il rimanente output ed esci

L'istruzione **apply-templates**, ha come attributo **select**. **Select** è uno degli attributi più importanti di XSL e consente di puntare ad un determinato elemento del file di origine attraverso la sintassi di XPath.

Applicazione di template nominati

Vediamo adesso lo stesso foglio di stile scritto utilizzando gli elementi nominati invece di quelli legati agli elementi:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
    <head>
    <title>Esempio Hello World</title>
    </head>
    <body>
    <xsl:call-template name="write-titolo"></xsl:call-template>
    <xsl:call-template name="write-messaggio"></xsl:call-template>
    </body>
</html>
</xsl:template>
<xsl:template name="write-titolo">
    <h1><xsl:value-of select="helloworld/titolo"/></h1>
</xsl:template>
<xsl:template name="write-messaggio">
        <div>
            <i>
            <xsl:value-of select="helloworld/messaggio"/>
            </i>
        </div>
</xsl:template>
</xsl:stylesheet>
```

Possiamo vedere come sia diverso il modo di richiamare il template : **call-template** infatti non si basa più su una selezione di un elemento del file XML di origine bensì sul nome stesso del template.

I template di questo tipo sono più simili a metodi e funzioni dei linguaggi di programmazione tradizionali, tanto che possono accettare anche parametri e non essere legati alla lettura del file XML originale.

Sui template nominati torneremo più quando prenderemo in esame:

- Parametri, variabili e loro scope
- Elementi per l'iterazione <xsl:for-each>
- Strutture condizionali <xsl:if> e <xsl:when>

6.3.1 Le variabili

In ogni linguaggio di programmazione una variabile corrisponde ad un "contenitore" di dati di varia natura.

Prendiamo questo passaggio di pseudo-codice:

```
a=1
b=a + 2
```

Avremmo così creato una variabile (contenitore) "a" assegnandogli il valore 1 e successivamente creiamo una nuova variabile "b" al quale viene assegnato il valore corrispondente alla somma di "a" e del valore 2 (b assumerà quindi il valore di 3)

La variabile si chiama così perché il valore che gli viene assegnato può variare nel corso del programma:

```
a=1 // qui a vale 1
b=a + 2
a=b // da qui a vale 3
```

Nel linguaggio di trasformazione XSL la dichiarazione e la gestione delle variabili, come vedremo successivamente, è sensibilmente diversa rispetto ad un linguaggio di programmazione tradizionale.

Dichiarazione di variabili

Possiamo dichiarare una variabile attraverso due elementi del linguaggio XSL : l'elemento *<xsl:param>* e l'elemento *<xsl:variable>*. Partiamo subito da un esempio per comprendere meglio il meccanismo.

Prediamo un file XML di esempio:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="variabili.xsl"?>
<indirizzi>
    <indirizzo>
        <id>1</id>
        <nome>Antonio</nome>
        <cognome>Rossi</cognome>
        <via>G.Verdi, 3</via>
        <comune>Roma</comune>
    </indirizzo>
    <indirizzo>
        <id>2</id>
        <nome>Giuseppe</nome>
        <cognome>Bianchi</cognome>
        <via>G.Rossini, 4</via>
        <comune>Milano</comune>
    </indirizzo>
</indirizzi>
```

Definiamo quindi il foglio di stile nel file nominato come **variabili.xsl** posto nella stessa directory:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <!-- dichiaro una variabile parametro id valida in    tutto il contesto
                                                                        -->
    <xsl:param name="id">2</xsl:param>
        <xsl:template match="/">
        <xsl:call-template name="stampa-nomi"></xsl:call-template>
    </xsl:template>
        <xsl:template name="stampa-nomi">
        <table border="1">
            <xsl:for-each select="//indirizzo[id=$id]">
                <tr>
                    <td>
                        <xsl:value-of select="nome"/>
                    </td>
                    <td>
                        <xsl:value-of select="cognome"/>
                    </td>
                    <td>
                        <xsl:value-of select="via"/>
                    </td>
                    <td>
                        <xsl:value-of select="comune"/>
                    </td>
                </tr>
            </xsl:for-each>
        </table>
    </xsl:template>
</xsl:stylesheet>
```

Notiamo che con l'elemento *<xsl:param>* abbiamo definito una variabile che prende il nome dal valore dell'attributo **name** (in

questo caso "*id*") ed ha il valore impostato nel contenuto dell'elemento. Da notare che è possibile impostare il valore della variabile anche attraverso l'attributo **select** dell'elemento *<xsl:param>*, in questo modo:

```
<xsl:param name="id" select="2"/>
```

Questa variabile è dichiarata come *Globale* rispetto al foglio di stile perché è esterna agli elementi *<xsl:template>* e posta sotto l'elemento radice *<xsl:stylesheet>*. Ciò in pratica significa che sarà possibile utilizzarne il valore in ogni parte del documento.

La variabile si utilizza anteponendo al nome il simbolo **$** come è stato fatto nella riga :

```
<xsl:for-each select="//indirizzo[id=$id]">
```

Differenze nella gestione delle variabili tra XSL e programmazione tradizionale

Dall'esempio che abbiamo visto emergono già le differenze nella gestione delle variabili tra XSL ed i linguaggi di programmazione ai quali siamo abituati.

In XSL infatti:

- Le variabili vengono valorizzate **unicamente** nel momento stesso della loro dichiarazione
- Non esiste alcun costrutto che consenta di variare il contenuto della variabile in altre parti della pagina (come ad esempio **$id=3** o simili)

In realtà in XSL le variabili non sono per niente ... *variabili* ma assomigliano piuttosto a quelle che nei linguaggi classici di programmazione sono chiamate costanti. Anche se sarebbe più cor-

retto definirle come dei *riferimenti*.
Si potrebbe obiettare: ma a cosa servono variabili di questo tipo? Nell'esempio non avremmo potuto usare direttamente il valore **2** anziché assegnarlo ad una variabile per poi richiamare quest'ultima?

Questo dubbio può venire perché nell'esempio abbiamo utilizzato una variabile che contiene un valore assoluto (il valore **2** appunto) , ma in XSL le variabili possono contenere anche un riferimento ad un elemento del documento XML di partenza (o, come vedremo in seguito, anche ad altri documenti XML esterni).

Vediamo infatti come potremmo utilizzare una variabile parametro per impostare un riferimento ad un elemento del documento di partenza:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <!-- dichiaro una variabile parametro come riferimento ad un
                                elemento e valida in tutto il contesto -->
    <xsl:param name="nodo-selezionato"
                                select="//indirizzo[id=2]"></xsl:param>
        <xsl:template match="/">
        <xsl:call-template name="stampa-nomi">
                                                </xsl:call-template>
    </xsl:template>
    <xsl:template name="stampa-nomi">
        <table border="1">
            <xsl:for-each select="$nodo-selezionato">
                <tr>
                    <td>
                        <xsl:value-of select="nome"/>
```

```
                    </td>
                    <td>
                        <xsl:value-of select="cognome"/>
                    </td>
                    <td>
                        <xsl:value-of select="via"/>
                    </td>
                    <td>
                        <xsl:value-of select="comune"/>
                    </td>
                </tr>
            </xsl:for-each>
        </table>
    </xsl:template>
</xsl:stylesheet>
```

Notiamo che adesso nell'attributo **select** di *<xsl:param>* c'è un'espressione che indica un riferimento ad un nodo con sintassi XPath. A questo punto potremo utilizzare la variabile appunto come alias o riferimento al nodo che rappresenta come in :

```
<xsl:for-each select="$nodo-selezionato">
```

È importante capire questo concetto di *riferimento* che assumono le variabili in XSL perché può tornare utile in molte occasioni.

Ambito e tipi di variabili

Torniamo adesso sulla dichiarazione delle variabili. Negli esempi precedenti abbiamo visto dichiarazioni di variabili con l'elemento *<xsl:param>* a livello dell'intero foglio di stile.

Le variabili però possono essere dichiarate anche all'interno di un singolo *<xsl:template>* così:

```
<xsl:template name="stampa-nomi">
<!-- dichiaro una variabile parametro id valida solo
                                        in questo template -->
   <xsl:param name="id">2</xsl:param>
      <table border="1">
         <xsl:for-each select="//indirizzo[id=$id]">
      ecc...
</xsl:template>
```

In questo modo il riferimento alla variabile è valido solo nell'ambito del template nel quale è stata dichiarata. Il tentativo di fare riferimento ad una variabile fuori dal suo ambito genererà pertanto un errore.

Abbiamo detto all'inizio che le variabili possono essere dichiarate anche attraverso l'elemento *<xsl:variable>*. In molti casi dichiarare una variabile con *<xsl:param>* o con *<xsl:variable>* è sostanzialmente equivalente.

Cioè, scrivere:

```
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:param name="id">2</xsl:param>
```

oppure

```
<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:variable name="id">2</xsl:variable>
```

è la stessa cosa.
Come la variabile *<xsl:param>* anche quella dichiarata con

<xsl:variable> ha un ambito differente a seconda che venga posta sotto l'elemento *<xsl:stylesheet>* o all'interno di un elemento *<xsl:template>*.

L'analogia però è soltanto apparente. In realtà *<xsl:param>* e *<xsl:variable>* hanno ruoli diversi anche se a volte vengono utilizzati indifferentemente.

Come suggerisce anche il nome *<xsl:param>* sta a rappresentare un parametro e come tale si comporta:

- se dichiarato a livello di foglio di stile (posto sotto l'elemento *<xsl:stylesheet>*) diventa un **parametro globale**.
- se dichiarato a livello di procedura (posto sotto l'elemento *<xsl:template>*) diventa un **parametro di procedura.**

Parametri globali

I parametri globali sono valori validi e richiamabili nell'ambito dell'intero foglio di stile.

In apparenza sono del tutto identici ad analoghe dichiarazioni che utilizzino *<xsl:variable>* anziché *<xsl:param>*. In realtà quasi tutte le librerie di trasformazione di XSL dispongono di metodi per impostare il valore dei parametri globali prima di compiere la trasformazione stessa. In questo modo otteniamo un valore dinamico e non più statico.

Parametri di procedura

Un ruolo del tutto diverso giocano invece i parametri di procedura, ovvero quelli dichiarati sotto l'elemento *<xsl:template>*. Per comprenderlo torniamo esaminiamo questo foglio di stile:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
```

```
                        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:call-template name="stampa-nomi">
            <xsl:with-param name="id" select="1"/>
        </xsl:call-template>
    </xsl:template>
        <xsl:template name="stampa-nomi">
      <xsl:param name="id">2</xsl:param>
        <table border="1">
            <xsl:for-each select="//indirizzo[id=$id]">
                <tr>
                    <td>
                        <xsl:value-of select="nome"/>
                    </td>
                    <td>
                        <xsl:value-of select="cognome"/>
                    </td>
                    <td>
                        <xsl:value-of select="via"/>
                    </td>
                    <td>
                        <xsl:value-of select="comune"/>
                    </td>
                </tr>
            </xsl:for-each>
        </table>
    </xsl:template>
</xsl:stylesheet>
```

Applicando questo foglio di stile noteremo che la trasformazione viene applicata al nodo con id=1.
Questo perché, nella forma di *parametro* di *procedura*, l'elemento *<xsl:param>* diventa un argomento impostabile in fase di

chiamata (con l'elemento *<xsl:with-param>*) all'interno di <**xsl:call-template**>.
Per il lettore con un background di programmazione tradizionale diremmo che un parametro di procedura è l'equivalente dei parametri di un metodo.
In javascript, per esempio, potremmo scrivere :

```
function Stampa_nomi (id) {
    //istruzioni....
}
```

E richiamare il metodo con

```
Stampa_nomi(1);
```

Ma torniamo a XSL sottolineando come attraverso i parametri di procedura possiamo creare delle unità di trasformazione riutilizzabili e parametrizzate.

Nel caso specifico il template "stampa-nomi" avrà output differenti a seconda se viene richiamato con:

```
<xsl:call-template name="stampa-nomi">
<xsl:with-param name="id" select="1"/>
</xsl:call-template>
```

o con:

```
<xsl:call-template name="stampa-nomi">
<xsl:with-param name="id" select="2"/>
</xsl:call-template>
```

Avrete anche notato come nella dichiarazione il parametro sia stato dotato di un valore iniziale 2:

```
<xsl:param name="id">2</xsl:param>
```

È per questo che se richiamiamo il template "stampa-nomi" senza parametri con

```
<xsl:call-template name="stampa-nomi"/>
```

Il parametro assume il valore di **default** ed il template produrrà comunque un risultato.

6.3.2 Istruzioni condizionali

Ogni linguaggio di programmazione non può prescindere da costrutti che consentano di definire delle condizioni. Anche XSL (pur non essendo ripetiamo un vero e proprio linguaggio di programmazione ma piuttosto un insieme di regole dichiarative destinate ad essere interpretate) non sfugge alla regola.

Gli elementi che esprimono le istruzioni condizionali sono:

```
∑ <xsl:if>
∑ <xsl:choose>
```

if

L' elemento *<xsl:if>* è un semplice **if** ad una sola condizione. L'espressione da verificare è contenuta dell'attributo **test** , ad esempio :

```
    <xsl:if test="name='rossi' ">
        Il nome è Rossi
    </xsl:if>
```

L'attributo **test** contiene la condizione da verificare nel documento di origine. Se l'espressione in questo attributo è **vera** il contenuto di *<xsl:if>* è inserito nell'output.
L'espressione valutata dall'attributo **test** può essere un Nodo del

documento xml nel contesto corrente o una comparazione tra stringhe o numeri, da notare che le stringhe all'interno delle espressioni si scrivono tra gli apici singoli; il valore '**rossi**' è una stringa mentre senza apici **rossi** sarebbe il riferimento al nodo **rossi** del documento di origine. I numeri invece si esprimono senza apici.

<u>*choose*</u>

<xsl:choose> è concettualmente simile al *Select* di visual basic e si compone di due sotto-elementi :

- *<xsl:when>* che contiene l'attributo "test" che rappresenta la condizione da verificare.
- *<xsl:otherwise>* che rappresenta il caso in cui nessuna delle condizioni espresse dagli elementi *<xsl:when>* si sia verificata.

Com'è facilmente intuibile *<xsl:choose>* deve contenere almeno un sotto-elemento *<xsl:when>* (altrimenti non ci sarebbe nessuna condizione da verificare). Com'è altrettanto ovvio che vi possa essere soltanto un sotto elemento *<xsl:otherwise>* mentre non vi sono limiti ai *<xsl:when>* da utilizzare.

Ma passiamo subito ad un esempio di *<xsl:choose>* :

```
<xsl:choose>
        <xsl:when test="name='Antonio'">
            Benvenuto Antonio
        </xsl:when>
        <xsl:when test="name='Giovanni'">
            Benvenuto Giovanni
        </xsl:when>
        <xsl:otherwise>
            Benvenuto chiunque tu sia
        </xsl:otherwise>
```

```
  </xsl:choose>
```

Nella prima condizione viene verificata la corrispondenza che contenuto del nodo <*name*> al valore letterale 'Antonio' e prodotto un certo output, nella seconda si verifica la corrispondenza rispetto al valore 'Giovanni' ed infine si fornisce una soluzione di Default.

Operatori di comparazione delle istruzioni condizionali
Gli operatori di comparazione sono:

1. "=" Operatore di eguaglianza , uguale a . Applicabile a stringhe e a numeri.
   ```
   <xsl:if test="name='rossi' ">
   ```
   valuta se il testo contenuto nel nodo **name** è uguale alla stringa 'rossi'
   ```
   <xsl:if test="id= 0 ">
   ```
   valuta se il testo contenuto nel nodo **id** interpretato come numero è uguale al numero 0

2. "!=" Operatorie di disuguaglianza, diverso da. Applicabile a stringhe e a numeri
   ```
   <xsl:if test="name!='rossi' ">
   ```
   valuta se il testo contenuto nel nodo **name** è diverso dalla stringa 'rossi'
   ```
   <xsl:if test="id!= 0 ">
   ```
   valuta se il testo contenuto nel nodo **id** interpretato come numero è diverso dal numero 0

Vi sono poi anche operatori di comparazione che operano solo sui numeri è sono :

- > (maggiore) e
- < (minore)

Tuttavia all'interno delle espressioni non possono essere indica-

ti direttamente perché, secondo la sintassi XML > e < non possono essere usati all'interno degli attributi. Si ricorre allora alle corrispondenti entità : *>*; per > e *< per* <.

Quindi avremo:

3. "**>**;" Operatore di confronto "maggiore". Applicabile solo a numeri.
   ```
   <xsl:if test="id &gt; 0 ">
   ```
   valuta se il testo contenuto nel nodo **id** interpretato come numero è maggiore del numero 0

4. "**<**;" Operatore di confronto "minore". Applicabile solo a numeri.
   ```
   <xsl:if test="id &lt; 10 ">
   ```
   valuta se il testo contenuto nel nodo **id** interpretato come numero è minore del numero 10

Naturalmente è possibile combinare gli operatori **>**; e **<**; con l'operatore = per ottenere un confronto maggiore/uguale a o minore/uguale a:

```
    <xsl:if test="id &lt;= 10 ">
```

valuta se il testo contenuto nel nodo id interpretato come numero è minore o uguale a 10

L'inserimento nell'espressione di test del solo nome di un nodo, invece, produce il valore true se il nodo esiste e false in caso contrario :

```
    <xsl:if test="name">
```

valuta se nel contesto corrente esistono uno o più nodi chiamati "name"

6.3.3 I Cicli

Nei linguaggi di programmazione classici si intende per ciclo un costrutto che consenta di ripetere una certa operazione più volte fino a che non si verifica una data condizione.

Non diverso è il concetto di ciclo in XSL dove è rappresentato dall'elemento *<xsl:for-each>*.

I cicli con *<xsl:for-each>* consentono di ripetere un determinato output per tutti i nodi individuati utilizzando l'attributo "select". Prendiamo ad esempio questo frammento di codice:

```
<table>
    <xsl:for-each select="indirizzi">
        <xsl:for-each select="indirizzo">
            <tr>
                <td>
                    <xsl:value-of select="nome"/>
                </td>
                <td>
                    <xsl:value-of select="cognome"/>
                </td>
                <td>
                    <xsl:value-of select="via"/>
                </td>
                <td>
                    <xsl:value-of select="comune"/>
                </td>
            </tr>
        </xsl:for-each>
    </xsl:for-each>
</table>
```

Qui abbiamo due cicli nidificati, il primo legge tutti i nodi <indi-

rizzi> del documento XML, quando trova un nodo passa il controllo al secondo ciclo che, all'interno del nodo *<indirizzi>*, legge tutti i sottonodi *<indirizzo>*. Il secondo ciclo può quindi accedere ai sotto-nodi di *<indirizzo>* che vengono inseriti all'interno di un output HTML standard.

È importante sottolineare come all'interno di un ciclo *<xsl:for-each>* cambia il *contesto* di lettura cioè come tutti i riferimenti a nodi siano relativi al nodo correntemente esaminato per cui il riferimento al nodo **nome** nel contesto del secondo ciclo del nostro esempio sarà corrispondente a **indirizzi/indirizzo/nome**.

Naturalmente nell'output ci può essere bisogno di far riferimento a un nodo posto al di fuori del contesto corrente. Consideriamo questo documento XML :

```
<province>
    <provincia>
        <sigla>MI</sigla>
        <comune>
            <nome>Milano</nome>
        </comune>
    </provincia>
    <provincia>
        <sigla>ROMA</sigla>
        <comune>
            <nome>Roma</nome>
        </comune>
    </provincia>
</province>
```

Poniamo che da questo documento dovessimo ricavare una semplice tabella HTML che mostri il nome del comune con accanto la sigla della sua provincia. Esprimendo il tutto con una serie di *<xsl:for-each>* potremmo scrivere :

```
<table>
    <xsl:for-each select="province">
        <xsl:for-each select="provincia">
            <xsl:for-each select="comune">
            <tr>
                <td>
                    <xsl:value-of select="nome"/>
                </td>
                <td>
                    <xsl:value-of select="../sigla"/>
                </td>
            </tr>
            </xsl:for-each>
        </xsl:for-each>
    </xsl:for-each>
</table>
```

Notiamo che sotto il ciclo "comune" dobbiamo leggere il nodo "sigla" che si trova sotto il nodo superiore <*provincia*> e quindi non è figlio (child) di <*comune*> ma è come lui figlio di <*provincia*>.

Abbiamo fatto ricorso dell'espressione XPath "**../sigla**" per selezionare il nodo sigla a partire dal contesto relativo al nodo superiore a quello corrente. XPath offre una grande flessibilità per la selezione di nodi, attributi e quant'altro basti pensare a come sarebbe stato possibile esprimere l'esempio precedente utilizzando pienamente XPath:

```
<table>
    <xsl:for-each select="province/provincia/comune">
            <tr>
                <td>
```

```
                    <xsl:value-of select="nome"/>
                </td>
                <td>
                    <xsl:value-of select="../sigla"/>
                </td>
            </tr>
    </xsl:for-each>
</table>
```

Risparmiamo cioè due cicli *<xsl:for-each>* semplificando di molto la sintassi e la leggibilità.

6.3.4 Ordinamento dei nodi

Complemento, a volte indispensabile, dell'elemento *<xsl:for-each>* è l'elemento *<xsl:sort>*.
Come suggerisce il nome *<xsl:sort>* si occupa di ordinare i nodi recuperati con *<xsl:for-each>*.
Come sempre partiamo prima da un esempio per comprendere meglio.

Prendiamo in considerazione una sorgente di dati Xml, una semplice lista di indirizzi :

```
<indirizzi>
    <indirizzo>
        <id>1</id>
        <nome>Antonio</nome>
        <cognome>Rossi</cognome>
        <via>G.Verdi, 3</via>
        <comune>Roma</comune>
    </indirizzo>
    <indirizzo>
        <id>2</id>
```

```
        <nome>Giuseppe</nome>
        <cognome>Bianchi</cognome>
        <via>G.Rossini, 4</via>
        <comune>Milano</comune>
    </indirizzo>
</indirizzi>
```

Poniamo il caso che, nella trasformazione, sia necessario ordinare prima i record in ordine alfabetico per cognome.
Applichiamo quindi la trasformazione utilizzando il seguente foglio di stile:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
              xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
    <xsl:for-each select="//indirizzo">
        <xsl:sort select="cognome" order="ascending" data-
                                                type="text"/>
        <div>
        <xsl:value-of select="nome"/> <xsl:value-of
                                          select="cognome"/>
        </div>
    </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

La lettura a questo punto dovrebbe essere abbastanza agevole:

1. Per mezzo di un ciclo *<xsl:for-each>* andiamo a selezionare tutti (si noti la path "//") i nodi che hanno nome "indirizzo" indipendentemente dalla loro posizione.
2. Immediatamente sotto *<xsl:for-each>* inseriamo un elemento *<xsl:sort>* per presentare i nodi secondo l'ordinamento de-

finito dagli attributi : select (nodo o attributo da prendere in considerazione per l'ordinamento), order (tipo di ordinamento; ascendente o discendente) e data-type (tipo di dati; testo o numeri).

3. Inseriamo, all'interno del ciclo *<xsl:for-each>* e sotto *<xsl:sort>* la logica di trasformazione.

Il risultato della trasformazione sarà appunto :

- Giuseppe Bianchi
- Antonio Rossi

Gli attributi dell'elemento <xsl:sort>

L'ordinamento viene impostato mediante gli attributi di <**xsl:sort**>.

select – È l'attributo più importante che definisce, in linguaggio XPath, quale espressione utilizzare nel confronto per ordinare due nodi e stabilire la priorità. Indica in pratica dove risiedono i dati da ordinare: in un altro nodo(solitamente più interno), in un attributo , ecc...

Le possibilità di definire dove stanno i dati da utilizzare nel confronto sono praticamente infinite e sono date dalla flessibilità di XPath, avendo comunque a mente che l'espressione da scrivere nel select di *<xsl:sort>* è sempre relativa rispetto al nodo corrente di *<xsl:for-each>*.

Solo qualche esempio:

<xsl:sort select="../@categoria"/>	usa il valore dell'attributo "categoria" del nodo superiore rispetto a quello corrente
<xsl:sort select="@id"/>	usa il valore dell'attributo "id" del nodo corrente

<xsl:sort select="nome/@id"/>	usa il valore dell'attributo "id" del nodo "nome" contenuto in quello corrente
<xsl:sort select="substring (cognome,1,3)"/>	usa solo le prime 3 lettere del valore del nodo "cognome" contenuto in quello corrente

order – definisce l'ordine da applicare alla lista dei nodi e ammette solo i valori: *ascending* (ascendente) o *descending* (discendente).

data-type – indica il tipo di dati da ordinare e supporta soltanto i valori : *text* (ordinamento alfanumerico) e *number* (ordinamento numerico).

6.3.5 Gli elementi <xsl:attribute> e <xsl:attribute-set>

L'elemento *<xsl:attribute>* serve per creare un attributo da inserire nell'elemento di output corrente.
Ad esempio questo pezzo codice :

```
...
	<table>
		<xsl:attribute name="border">0</xsl:attribute>
		<xsl:attribute name="cellpadding">0</xsl:attribute>
		<xsl:attribute name="width">100%</xsl:attribute>
		<tr>
			<td></td>
		</tr>
	</table>
...
```

Produrrà, come output il seguente :

```
<table border="0" cellpadding="0" width="100%">
```

```
   <tr>
      <td>
      </td>
   </tr>
</table>
```

Notiamo quindi come l'elemento *<xsl:attribute>* definisca l'attributo contenuto in name all'interno dell'elemento di output nel quale è inserito.

La cosa interessante è che gli attributi si possono raccogliere in "collezioni" mediante l'attributo *<xsl:attribute-set>* da inserire sotto il nodo radice del foglio di stile.

Nel caso precedente avremmo quindi potuto raccogliere tutti gli attributi in un elemento *<xsl:attribute-set>* in questo modo:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:attribute-set name="table-def">
     <xsl:attribute name="border">0</xsl:attribute>
     <xsl:attribute name="cellpadding">0</xsl:attribute>
     <xsl:attribute name="width">100%</xsl:attribute>
</xsl:attribute-set>
...
</xsl:stylesheet>
```

In questo modo nel codice di output sarebbe stato sufficiente inserire il nome attribuito al set di attributi in questo modo:

```
...
<table xsl:use-attribute-sets="table-def">
```

```
    <tr>
        <td></td>
    </tr>
</table>
...
```

xsl:use-attribute-sets (unico attributo definito nello schema XSL) permette infatti di connettere ad un elemento di output una lista di attributi definita con nome.
In questo modo è possibile definire più liste di attributi per i nodi da impiegare in parti diverse del codice di output migliorandone gestione e leggibilità.

6.3.6 Elementi di inclusione

La sintassi XSL prevede due tipi fondamentali di inclusione :

- L'inclusione di parti di codice XSL
- Il riferimento a sorgenti XML diverse dal documento principale oggetto di trasformazione

Entrambe le tipologie costituiscono un potente strumento per la razionalizzazione del codice.

\inclusione di parti di codice XSL
L'inclusione di parti di codice XSL assomiglia agli include presenti in ASP o PHP e consente di dividere il codice XSL in files separati.

Questa pratica presenta due grossi vantaggi :

1. riutilizzo delle stesse funzioni in fogli di stile diversi
2. miglioramento della gestione e della leggibilità del codice

È possibile ad esempio mettere in un foglio di stile separato alcuni parametri (colore, caratteri ecc...) utilizzati nella trasformazione in modo da poterli cambiare in un unico punto anziché andarli a ricercare in tutti i fogli di stile.

L'inclusione di parti di codice può essere effettuata tramite due elementi :

- *<xsl:import>*
- *<xsl:include>*

Entrambi hanno la stessa sintassi con un solo attributo *href* costituito dall'url relativo o assoluto del secondo foglio di stile da includere.

La differenza fondamentale tra *<xsl:import>* e *<xsl:include>* è che nel primo tutte le regole e i template definiti nel foglio di stile importato hanno la precedenza (nel caso di nome uguale) rispetto a quelli del foglio di stile corrente. Per questo *<xsl:import>* deve essere inserito come primo elemento sotto il nodo radice *<xsl:stylesheet>* del foglio di stile mentre per *<xsl:include>* non è necessario rispettare tale ordine.

Passiamo quindi a vedere nella pratica l'utilizzo di un <**xsl:include**>.

Come sorgente XML utilizziamo la lista di indirizzi che abbiamo visto in precedenza.

Impostiamo quindi il foglio di stile da includere che metteremo nella stessa directory del foglio di stile principale e chiameremo *common.xsl* per far immediatamente comprendere che si tratta di codice che può essere utilizzato da più fogli di stile.

Inseriamo in *common.xsl* solo una collezione di attributi applicabili ad una tabella HTML:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
```

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:attribute-set name="table-def">
        <xsl:attribute name="border">0</xsl:attribute>
        <xsl:attribute name="cellpadding">0</xsl:attribute>
        <xsl:attribute name="width">100%</xsl:attribute>
    </xsl:attribute-set>
</xsl:stylesheet>
```

Passiamo quindi a creare il foglio di stile principale da applicare alla sorgente XML definendo al suo interno un riferimento *<xsl:include>* a common.xsl e utilizzando, a questo punto, la collezione di attributi come se fosse stata definita all'interno del foglio principale:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:include href="Common.xsl"/>
<xsl:template match="/">
<table xsl:use-attribute-sets="table-def">
<xsl:for-each select="//indirizzo">
<tr>
    <td><xsl:value-of select="nome"/></td>
    <td><xsl:value-of select="cognome"/></td>
    <td><xsl:value-of select="via"/></td>
    <td><xsl:value-of select="comune"/></td>
</tr>
</xsl:for-each>
</table>
</xsl:template>
</xsl:stylesheet>
```

Ai veterani dei linguaggi di scripting per il web come ASP o PHP non

sarà sfuggita l'utilità della possibilità di inclusione per garantire la modularità dei fogli di stile.

riferimento a sorgenti XML esterne
Ugualmente preziosa, anche se per altri motivi, è la possibilità di far riferimento a sorgenti XML esterne rispetto al documento che stiamo trasformando.
Il riferimento a sorgenti esterni si realizza attraverso una funzione del tutto simile alle funzioni XPath: la funzione document().

6.3.7 La funzione document()

La funzione document() è molto versatile perché restituisce oggetti diversi a seconda del numero di parametri utilizzati:

1. **document**('doc.xml') - Utilizzando un solo parametro questo viene inteso come URL e l'oggetto restituito sarà il documento XML corrispondente come set di nodi.
2. **document**('doc.xml', espressione) – Utilizzando due parametri il primo viene inteso come URL ed il secondo come il set di nodi a cui fare riferimento e viene restituito quest'ultimo.
3. **document**() – Utilizzando la funzione senza parametri viene restituito il foglio di stile stesso come documento XML.

Ma come utilizzare in pratica questa possibilità?

Poniamo di avere il nostro solito file XML degli indirizzi ma di aver attribuito ad ogni indirizzo una categoria sotto forma di codice:

```
<?xml version="1.0"?>
<indirizzi>
    <indirizzo>
        <id>1</id>
        <nome>Antonio</nome>
        <cognome>Rossi</cognome>
```

```
            <via>G.Verdi, 3</via>
            <comune>Roma</comune>
            <categoria>2</categoria>
      </indirizzo>
      <indirizzo>
            <id>2</id>
            <nome>Giuseppe</nome>
            <cognome>Bianchi</cognome>
            <via>G.Rossini, 4</via>
            <comune>Milano</comune>
            <categoria>1</categoria>
      </indirizzo>
</indirizzi>
```

Mettiamo quindi di aver definito le categorie in un secondo file XML (*categorie.xml*) :

```
<?xml version="1.0"?>
<categorie>
      <categoria cod="1">Lavoro</categoria>
      <categoria cod="2">Privato</categoria>
</categorie>
```

Come facciamo a collegare le descrizioni contenute nel secondo file ai nominativi contenuti nel primo file?

Risolviamo il problema con questo foglio di stile:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="categorie" select="document('categorie.xml')"/>
      <xsl:template match="/">
```

```
            <table>
                  <xsl:for-each select="//indirizzo">
                        <tr>
                              <td><xsl:value-of select="nome"/></td>
                              <td><xsl:value-of select="cognome"/></td>
                              <td><xsl:value-of select="via"/></td>
                              <td><xsl:value-of select="comune"/></td>
                              <td>
      <xsl:value-of select="$categorie//categoria[@cod=
                                                current()/categoria]"/>
                              </td>
                        </tr>
                  </xsl:for-each>
            </table>
      </xsl:template>
</xsl:stylesheet>
```

In pratica assegniamo il node-set prodotto dalla lettura del secondo file XML ad un parametro "categorie" che viene poi utilizzato nel ciclo *<xsl:for-each>*. Quando si tratta di estrarre il valore della categoria relativa all'indirizzo usiamo la sintassi XPath per trovare, nel node-set, il nodo categoria che abbia l'attributo *cod* corrispondente al valore del nodo "categoria" posto sotto al nodo "indirizzo" corrente. Da notare l'utilizzo della funzione XSL *current()* che sta ad indicare il nodo corrente della funzione *<xsl:for-each>* che serve per evitare ambiguità nel riferimento ai nodi.

6.4 CONCLUSIONI

Abbiamo visto in questo capitolo la tecnologia di trasformazione XSL.
Il linguaggio offre anche una serie di altre funzioni "minori" e un'infinita gamma di "trucchi" a disposizione del programmatore.
C'è da dire che altre funzioni, molto interessanti sono offerte dalle

possibilità di estensione al linguaggio offerte dalle varie piattaforme di implementazione (.NET, PHP, Java ecc...).
In questa sede però ho cercato di attenermi il più possibile a sintassi e funzioni standard, che possono essere utilizzate in ogni contesto.
In alcune applicazioni (Web ma non solo) XSL costituisce uno dei fondamenti per costruire dei framework personali anche molto ricchi e complessi.

RIFERIMENTI

Links

Il sito di riferimento per lo standard XML e le tecnologie collegate trattate nel libro è quello del W3C (World Wide Web Consortium), *http://www.w3.org* , che si occupa di definire e aggiornare gli standard.

Un altro sito è *http://www.w3schools.com* dove si trovano interessanti tutorials su XPath, DOM e XSL.

Per i programmatori Windows, non può mancare poi il riferimento alla Library, *http://msdn.microsoft.com/library* , dove si trovano tutte le informazioni per utilizzare XML con le tecnologie Microsoft.

Anche Java, ovviamente, rivolge particolare attenzione allo standard XML. Tutorials, API e quant'altro li potete trovare in *http://java.sun.com/xml.*

Su *http://www.xml.com* si trovano sempre dei tutorials anche per l'uso di XML con altri linguaggi come PHP, Perl e Python.

Altra risorsa preziosa, specialmente per XSL, è il sito *http://www.topxml.com* dov'è trattata a fondo la connessione tra la tecnologia .NET e XML.

Su *http://xml.html.it* poi si trovano guide su XML anche in lingua italiana.

Libri

Per i libri in italiano facciamo riferimento alla lista pubblicata da *http://www.techbook.it*:

Titolo	**Editore**
.NET XML & Web Services Full Contact	Mondadori Informatica

Costruire siti Web con XML	Tecniche Nuove
Creare XML Web Services	Mondadori Informatica
HTML & XML Passo per Passo	Mondadori Informatica
HTML e XML per principianti	Mondadori Informatica
Imparare XML in 24 ore	Tecniche Nuove
Imparare XML in 6 ore	Tecniche Nuove
Programmare BizTalk Server con XML e SOAP	Mondadori Informatica
Programmare Microsoft SQL Server 2000 con XML	Mondadori Informatica
Programmare Microsoft SQL Server 2000 con XML, Seconda Edizione	Mondadori Informatica
Programmare Web Services con XML-RPC	Hops Libri
Programmare XML	Mondadori Informatica
Programmare XML in Microsoft .NET	Mondadori Informatica
SOAP Guida all'uso	Mondadori Informatica
Sviluppare XML Web Services e componenti Server con Visual Basic .NET e C#.NET MCAD/MCSD Training	Mondadori Informatica
Sviluppo di soluzioni XML	Mondadori Informatica
XML	Tecniche Nuove
XML	McGraw-Hill Informatica
XML corso di programmazione	Apogeo
Xml e Java	McGraw-Hill Informatica
Xml Espresso For Dummies	Apogeo
XML Guida Completa	Apogeo
Xml Guida Di Riferimento	Apogeo
XML Guida tascabile	Mondadori Informatica
XML I Portatili	Mondadori Informatica
XML La Guida Completa	McGraw-Hill Informatica
XML Passo per Passo	Mondadori Informatica
XML Passo per Passo, Seconda edizione	Mondadori Informatica
XML Pocket Reference	Hops Libri
XML Schema	McGraw-Hill Informatica
Xml Tutto & Oltre	Apogeo
XML – Le basi	Tecniche Nuove
XSLT Guida Completa	Apogeo

i libri di

COMPRENDERE XML

Autore: Francesco Smelzo

EDITORE
Edizioni Master S.p.A.
Sede di Milano:Via Ariberto, 24 - 20123 Milano
Sede di Rende: C.da Lecco, zona ind. - 87036 Rende (CS)

Realizzazione grafica:
Cromatika Srl
C.da Lecco, zona ind. - 87036 Rende (CS)

Art Director: Paolo Cristiano
Responsabile grafico di progetto: Salvatore Vuono
Coordinatore tecnico: Giancarlo Sicilia
Illustrazioni: Tonino Intieri
Impaginazione elettronica: Francesco Cospite

Servizio Clienti
☎ **Tel. 02 831212 - Fax 02 83121206**
@ **e-mail: customercare@edmaster.it**

Stampa: Grafica Editoriale Printing - Bologna

Finito di stampare nel mese di Ottobre 2006